University of Alberta

Library Release Form

Name of Author: Kevin Robert Charter

Title of Thesis: A Constructive Type Theory for Simple Imperative Programming

Degree: Master of Science

Year this Degree Granted: 1999

University of Alberta

A Constructive Type Theory for Simple Imperative Programming

by

Kevin Robert Charter ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 1999

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Constructive Type Theory for Simple Imperative Programming** submitted by Kevin Robert Charter in partial fulfillment of the requirements for the degree of **Master of Science**

To my family

Abstract

Traditional constructive type theory was developed for the λ-calculus and is directly applicable only to functional languages or functional fragments of languages. Unfortunately, the most popular languages today are imperative and have destructive operations with side-effects. If constructive type theory is to provide a formal tool for writing programs in these languages, it must adapt to imperative programming. In this thesis, we develop a constructive type theory for a simple imperative language with variables and arrays and destructive updates on them. The type system is expressed using Girard's linear logic extended with an additional connective and a special storage modality after Reddy, and constructive interpretations of the quantifiers. A significant fragment of the system is implemented in MIZAR-I, a simple proof checker and editor, and the theory is demonstrated by partially deriving a Bubble-sort function.

Acknowledgements

Special thanks to my supervisor, Jim Hoover, for guidance, useful discussions, and many lunches, movies and other diversions. Thanks also to the other members of my committee, Piotr Rudnicki and Curtis Hrischuk, for sound advice and constructive criticism that helped improve this work. I extend my appreciation to the Natural Sciences and Engineering Research Council of Canada for scholarship support, and to the Department of Computing Science at the University of Alberta for providing a comfortable and friendly working environment.

John Buchanan, Paul Ferry, and the other inhabitants of the graphics lab provided amusing times too numerous to mention. I hope that bump on the head didn't hurt too much, Paul.

And thanks to Lisa, for support and understanding.

Contents

List of Figures

List of Tables

List of Symbols

$\wedge$	conjunction	3
$\vee$	disjunction	3
$\to$	implication	3
$\neg$	negation	3
$\forall$	universal quantifier	4
$\exists$	existential quantifier	4
$\vdash$	sequent turnstile	5
Γ	sequent variable	5
Δ	sequent variable	5
λ	function abstraction symbol	6
$E[x/y]$	substitute x for y in E	6
:	term and type separator	8
$W_=(s)$	self-equality witness	13
$W_s(a, b)$	substitution witness	13
Fst	existential projection	14
Snd	existential projection	14
$\top$	additive unit	23
$\bot$	additive co-unit	23
$\mathbf{1}$	multiplicative unit	23
$\mathbf{0}$	multiplicative co-unit	23
$\star$	multiplicative conjunction	23
$+$	multiplicative disjunction	23
$\sqcap$	additive conjunction	23
$\sqcup$	additive disjunction	23
$\multimap$	linear implication	23
$\sim$	linear negation	23
!	storage modality	23
?	costorage modality	23
Voidm	multiplicative unit term	26
Voida	additive unit term	26
Abort(x)	additive counit term	26
(s, t)	strict pair	26
$\langle s, t\rangle$	lazy pair	26
π_i	projection (additive disjunction)	26
κ_i	variant tag (multiplicative disjunction)	26
Store(s)	storage term	26
Get(s)	term for storage retrieval	26
Ignore(s)	term for storage discard	26
$\triangleright$	before	48
$\dagger$	regenerative modality	48
$[s; t]$	lazy sequence (term for $\triangleright$)	48

RStore(s)	regenerative storage term	48
RGet(s)	term for regenerative storage retrieval	48
RIgnore(s)	term for regenerative storage discard	48
RConvert(s)	term for storage conversion	48
$\Gamma[\bullet]$	hole	47
letemp	(immobile) temporary definition	52
A **var**	variable	64
A **array** $[n]$	array	68

Chapter 1

Introduction

For several years formal methods have been studied by computer science academics and to varying extents put in practice by computer science professionals. The practical objective of formal methods is to reduce the number of errors made in developing software and hardware. Here the word "error" may mean a problem with the specification or planning of the system, or a problem with the implementation itself. The hope has chiefly been to avoid the first kind of error, the kind that usually turns out to be the most expensive in terms of time and money to fix, and perhaps for this reason some of the most successful formal methods have been chiefly concerned with formal specification[1].

Formal specification can make an important contribution to improving software quality. A precise and mathematical statement of what a system must do reduces ambiguity and better ensures that everyone involved in the project has a clear understanding of what the software must do. Consequently there is less chance for misunderstanding and miscommunication among the planners and implementors. But there is still plenty of room for error in moving from a formal specification to the actual implementation. Frequently there is no rigorous way to check that an implementation actually satisfies its specification—this is usually checked by some combination of testing and informal argument.

Constructive type theory (CTT) potentially provides a formal method which makes the entire software development process, from specification to implementation, rigorous. Traditionally, CTT is a marriage of intuitionistic first-order logic and functional programming in which functions may be specified within the logic and then implementations *automatically* extracted from the reasoning. The marriage is accomplished by a mapping between propositions in the logic and terms in the programming language called the Curry-Howard isomorphism. Propositions serve as *types* for their terms, and terms serve as *witnesses* or

[1]For example, specification languages like Z, or methods like model checking, where a system is modeled by a finite state machine, and properties are proven about the model.

proofs for their propositions; type-checking corresponds to ensuring that propositions are combined according to the inference rules of the logic. The type system is very rich, because propositions can assert properties of terms. Consequently, CTT can be viewed as a programming language in which programs can not only be written directly, but can be derived from a specification.

But of course there are drawbacks to using CTT, some of them quite serious. The problems include:

1. First and foremost, formal reasoning in CTT is more cumbersome than programming first and using informal reasoning to argue that the program meets its specification. CTT derivations can be long and unwieldy, even for simple programs.

2. How a program is derived has a profound impact on its space and time complexity. This interplay is subtle, and the programmer cannot effectively use CTT without knowing intimately the relationship between the logic and the term language. Efficient code is no longer a matter of good algorithms and knowing the programming language well—there is an extra layer between the programmer and the language that can make efficiency difficult to obtain.

3. Even with the greatest attention to proof structure in pursuit of efficiency, there is a systematic inefficiency introduced by any CTT: computationally irrelevant terms. For example, the witness of an existentially-quantified formula,

   $$\exists x : A.P[x]$$

   where x is some object of type A and $P[x]$ is a statement of some property that x satisfies, is a pair (s, t), where s is the witness of x, and t is the witness of the property. Often only s has any computational significance. The potentially very large t will at best occupy considerable heap space (in a lazy implementation of the target calculus) and at worst will also be evaluated and discarded without ever contributing to the computation (in a strict implementation). Terms like t are called *computationally irrelevant*. The elimination of computational irrelevance is studied in [13].

4. Finally, the programming model of traditional CTT is limited to functional programming. This is not a theoretical difficulty, since functional languages are in principle as powerful as imperative[2] and other languages, but it does pose a practical difficulty. For

[2]For the moment, an imperative language is a language with assignment. This will be made more precise in the following sections.

CTT to be a useful formal method, programmers should be able to write programs in the languages that they use for real systems. The vast majority of widely-used programming languages have imperative features and several important functional languages are by no means purely functional, having imperative features for efficiency reasons.

When one considers that no formal method is ever used to build large systems in their entirety, but only critical parts of those systems, this last point takes on a new significance. To use CTT to build part of a system, the programmer must be able to easily integrate parts built with CTT and the parts built in traditional ways, and this is easiest if the programming language is the same in both cases, and there are few restrictions on the features of the language that may be used in the CTT components.

This thesis attempts to address the last issue: in particular, it attempts to define and implement a CTT system whose term language has variables (terms which change their value or meaning as the program executes).

1.1 A CTT Primer

This section reviews CTT for the reader, and establishes a framework for the following chapters. We review the features of a CTT described in detail in [27], and discuss using this system for building programs.

1.1.1 Preliminaries: Notation and the Sequent Calculus

Before describing CTT itself, we review first-order logic, sequent calculus, and the λ calculus. Readers already familiar with these notions may skip to the next subsection.

First-order logic (FOL) is an extension of predicate logic, whose basic logical connectives appear in Table 1.1. We assume that the usual interpretation of the connectives as boolean functions is already familiar to the reader. Propositions are names beginning with capital

Connective	Meaning
$\wedge$	conjunction (and)
$\vee$	disjunction (or)
$\rightarrow$	implication (if-then)
$\neg$	negation (not)

Table 1.1: FOL Connectives

letters (A, B, ...), and predicates are similar names followed by brackets and a list of ar-

guments ($P[x]$, $Loves[x, y]$, ...). The arguments to predicates are called *terms* or *function symbols*, and are written using lower-case letters (x, y, *giraffe*), possibly followed by parentheses with a list of argument terms ($f(x)$, $g(y, f(z))$). The function symbols traditionally represent objects in some universe of discourse. For example, the statement "Jane likes John's mother" might be formalized as $Likes(jane, mother(john))$, where $Likes$ represents the *likes* relation on people, $jane$ represents Jane, $john$ represents John, and $mother(john)$ John's mother. In addition to the traditional connectives, FOL adds two *quantifiers*, the universal quantifier ($\forall$) and the existential quantifier ($\exists$). The universal quantifier is used to make statements about all the objects in a universe of discourse, such as "everyone either dislikes John or likes his mother"

$$\forall x. \neg Likes[x, john] \vee Likes[x, mother(john)]$$

and the existential quantifier to assert the existence of some (anonymous) object with a particular property, such as "there is a person whom everybody likes"

$$\exists z. \forall x. Likes[x, z].$$

The *well-formed*, or syntactically legal, formulas of FOL obey the following grammar:

$$\begin{aligned} wff &::= quant\ var\ .\ wff \\ &\mid wff\ binary\ wff \\ &\mid \neg\ wff \\ quant &::= \forall\ \exists \\ binary &::= \wedge\ \vee \end{aligned}$$

The $\neg$ operator binds most tightly, followed by $\wedge$, $\vee$, $\rightarrow$, in that order. The quantifiers $\forall$ and $\exists$ have equal precedence; formulas containing several quantifiers are read left-to-right.

Using the familiar boolean function interpretations of the connectives and quantifiers, any well-formed formula can be assigned a boolean value ($true$, $false$) in the presence of a *valuation*, an assignment of boolean values to proposition symbols. Determining the value of a well-formed formula by applying the boolean functions associated with the connectives is called *evaluating* the formula. A well-formed formula is *valid* if and only if it evaluates to $true$ for every possible valuation.

The *sequent calculus* is a formal notation for inference rules and proofs in FOL. A *sequent* has the form

$$\Gamma \vdash \Delta$$

where Γ and Δ are possibly empty lists of well-formed formulas, known as the *left* and *right* sides of the sequent, respectively, and the intervening symbol is the *turnstile*. Γ and Δ are both sets of well-formed formulas. A sequent $\Gamma \vdash \Delta$ is *valid* if and only if in any valuation in which the conjunction of the formulas in Γ evaluates to *true*, the *disjunction* of the formulas in Δ evaluates to *true*. That is, assuming all the formulas in Γ are true, at least one of the formulas in Δ is true. For $\Gamma = f_1, \ldots, f_n$ and $\Delta = g_1, \ldots, g_m$, the sequent is equivalent to

$$\vdash (f_1 \wedge \ldots \wedge f_n) \rightarrow (g_1 \vee \ldots \vee g_m).$$

That is, in the absence of any assumptions, the implication on the right of the turnstile is valid (true for any valuation). One sequent that is always valid is the *trivial* sequent, of the form

$$A \vdash A,$$

or, equivalently,

$$\vdash A \rightarrow A$$

Trivial sequents are used to introduce base assumptions in proofs.

The sequent calculus also provides a notation for inference rules and inference rule applications. For example, the rule for introducing a conjunction is

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B}$$

The rule says that if assumptions Γ lead to proposition A, and assumptions Δ lead to proposition B, then the union of the assumptions, Γ, Δ, leads to the conjunction $A \wedge B$. The arguments of the rule appear above, and the result below, a horizontal line. Proof *trees* compose inference rules by occupying the argument positions of later rules with the results for earlier rules. For example, given sequents $\Gamma \vdash A$, $\Delta \vdash B$ and $\Omega \vdash C$ the derivation of $\Gamma, \Delta, \Omega \vdash (A \wedge B) \wedge C$ may be written

$$\frac{\dfrac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \Omega \vdash C}{\Gamma, \Delta, \Omega \vdash (A \wedge B) \wedge C}$$

where the rule for introducing conjunctions has been applied twice, the result of the first application used as the first argument to the second application. Assumptions flow from sequent to sequent through the application of inference rules. The final sequent in a proof has the goal formula on the right of the turnstile, and the complete set of assumptions that lead to it on the left. The inference rules of FOL are chosen so that, given valid sequents as

arguments, the results are valid. Making the argument sequents trivial in the above proof tree, $\Gamma = A$, $\Delta = B$, $\Omega = C$ and the result is $A, B, C \vdash (A \wedge B) \wedge C$. Assuming that each of A, B, and C is true, so is the conjunction $(A \wedge B) \wedge C$.

We postpone writing out more of the inference rules of FOL until after introducing the λ-calculus and the Curry-Howard isomorphism.

1.1.2 The λ-calculus

The λ-calculus is a notation for writing functions developed and studied by Alonzo Church in the 1920s, 30s and 40s. Although originally designed as an alternative formal framework for expressing mathematics, it also forms a theoretical underpinning for functional programming languages.

The grammar for the λ-calculus is simply

$$\mathit{lexp} ::= \mathit{var} \mid (\lambda\ \mathit{var}\ .\ \mathit{lexp}) \mid \mathit{lexp}\ \mathit{lexp}$$

λ-calculus expressions include variables (single lower-case letters, x, y, $\ldots$), λ abstractions (function definitions), and applications. A variable x is *bound* in an expression E if there is an enclosing λx; x is free in E if it is not bound. The λ calculus has three basic rules for *reduction*, or the evaluation of λ expressions[12]. Let A and E be expressions, and let $\Rightarrow$ represent the *reduction relation* on expressions; then the reduction rules are
α **reduction:**

$$(\lambda x.E) \Rightarrow (\lambda y.E[y/x]) \qquad \text{provided } y \text{ is not free in } E;$$

β **reduction:**

$$(\lambda x.E)A \Rightarrow E[A/x];$$

η **reduction:** function abstractions;

$$(\lambda x.Ex) \Rightarrow E \qquad \text{provided } x \text{ is not free in } E$$

α reduction is a renaming rule that allows function parameter names to be changed; the restriction that the new name y must not be free in E prevents accidental *capture* of an existing free variable by the argument (such a capture would change the meaning of the expression). β reduction substitutes an actual argument for a function's formal parameter. η reduction is an optimization that removes unnecessary abstractions; the rule is a bit counter-intuitive, but consider the β reduction of $(\lambda x.Ex)A$ for an arbitrary expression A:

$$(\lambda x.Ex)A \Rightarrow (Ex)[A/x] \Rightarrow EA \qquad \text{if } x \text{ is not free in } E$$

so we can, without affecting the meaning, simplify $(\lambda x.Ex)$ to E.

An expression to which neither the β or η rules apply is fully reduced, or in *normal form*. Reduction or evaluation is the repeated application of the β and η rules until a normal form is reached, or the reduction sequence *terminates*. Unfortunately, there are expressions for which not every reduction sequence terminates, and expressions which have no terminating reduction sequences at all. For example, consider

$$(\lambda x.xx)(\lambda x.xx)$$

The only applicable reduction is β reduction, but this produces exactly the same expression again, and so the reduction process does not terminate. We can view this expression as a simple (but useless) infinite loop. While not all λ expressions reduce to a normal form, the λ calculus does have the following property:

Theorem 1 (Church-Rosser) *For any expression A, if $A \Rightarrow^* B$ and $A \Rightarrow^* C$, then there exists D such that $B \Rightarrow^* D$ and $C \Rightarrow^* D$.*

The Church-Rosser property ensures that when an expression has a normal form, it is unique (up to renaming bound variables via α reduction); every terminating sequence of reduction rules arrives at that normal form.

1.1.3 Types and the Curry-Howard Isomorphism

In the λ-calculus, any expression can be used as an argument to any function. The pathological expression

$$(\lambda x.xx)(\lambda x.xx)$$

exploits this flexibility by applying a function to *itself*. Although expressions like the one above pose no problem to the reduction rules, they are inconvenient because evaluation is not guaranteed to terminate. Perhaps they can be made illegal in some systematic way.

One approach to eliminating these problem expressions is to regulate which expressions are legal arguments to each function by imposing a *type system*. A type system has two components: a syntax for types, and a set of rules for assigning types to expressions. For example, we could give types the grammar

$$\mathit{type} ::== \mathit{typeconst} \mid \mathit{type} \rightarrow \mathit{type}$$

where *typeconst* can be any element of some set of symbols $\{A_1, A_2, \ldots\}$. For each production in the λ-calculus grammar we need a corresponding type assignment rule. The types of

variables are *declared*, and so are, in a sense, assumptions. We'll borrow some notation from the sequent calculus for FOL and write the rule for variables as a trivial sequent,

$$x : \sigma \vdash x : \sigma$$

where the variable x is paired with its type using an intervening semicolon. In plain words, the rule says that assuming x is of type σ, we can (trivially) conclude x is of type σ. Generally, the left side of one of these "sequents" is a set of type declarations on variables (akin to the set of declarations at the start of a block in C or Pascal), subject to one restriction: a set of declarations Γ is well-formed if and only if every variable x appearing in Γ appears exactly once; that is, there is only one type defined for any variable. The right side of a sequent is an assertion about the type of some expression involving the variables on the left. Sequents are also called type *judgments.*

For λ abstractions, we have the rule

$$\frac{\Gamma, x : \sigma \vdash E : \tau}{\Gamma \vdash (\lambda x.E) : \sigma \rightarrow \tau}$$

That is, if given a set of declarations that includes $x : \sigma$ we can conclude that E has type τ, then we can abstract x to produce a function of type $\sigma \rightarrow \tau$. The function has a formal argument of type σ and produces a result of type τ. For function applications, we require an exact match between the type of the actual argument and the formal parameter:

$$\frac{\Gamma \vdash F : \sigma \rightarrow \tau \qquad \Delta \vdash E : \sigma}{\Gamma, \Delta \vdash FE : \tau}$$

Expressing the typing rules in such a precise way facilitates proving properties of the type system. For example, it is possible to prove that this set of typing rules either assigns no type or exactly one type to any expression in the λ calculus. Even more important, it can be shown that the set of λ-calculus expressions that can be assigned a type in this type system enjoys the following property[27]:

Theorem 2 (Confluence) *Any expression E that can be assigned a type in the above type system has a normal form, and every sequence of β and η reductions is a prefix of some sequence that terminates at that normal form.*

If we think of this typable subset of the λ-calculus together with this type system as a typed programming language, the confluence theorem says that the evaluation of a program in this language always terminates.

The Curry-Howard Isomorphism

Notice that if we erase the λ-calculus expressions from the above typing rules, we get the fragment of FOL that deals exclusively with assumptions and implication. This may simply be a happy coincidence, or there may be something significant in the correspondence between logical implication and function types. Intuitively, the formal statement $F : \sigma \to \tau$ says that F takes an example of a σ and produces an example of a τ. Conversely, we might interpret expressions as a kind of "evidence" that their types are valid: the existence of F shows that if σ is valid (has some evidence E) then τ is valid since it has evidence FE.

It turns out that this interpretation of logical propositions as types, and λ-calculus expressions as evidence for propositions, can be extended to other connectives in propositional logic if we first extend the λ-calculus with a small but useful collection of new constructions. The pairing of λ-calculus constructions and formulas from propositional logic is called the *Curry-Howard isomorphism*, and it forms the foundation of constructive type theory.

We assume we are given some finite collection $\{\tau_1, \ldots, \tau_n\}$ of *base types*, whose structure is not apparent within the logic. The base types are simply 0-nary predicate names. For each of the τ_i there is a collection of expressions in the λ-calculus which are assigned type τ_i. The expressions assigned τ_i are called the *witnesses* or *inhabitants* of τ_i. For example, we might support the base types listed in Table 1.2. The example witnesses we have listed here

Type Predicate	Meaning	Example Witnesses
bool	booleans	true, false
N	natural numbers	$0, 1, \ldots$
char	characters in some alphabet	'a', 'b', ...

Table 1.2: Examples of base types

are not λ expressions, but they can be represented by λ expressions in standard ways [12]. We will take the liberty of extending the λ-calculus as necessary with more conventional syntax for such base type expressions. We will adopt the boolean and natural number base types for our example CTT.

All types other than base types are expressed using well-formed formulas from FOL. In FOL each connective has one or more introduction rules, which allow a formula containing the connective to be built up from simpler formulas, and one or more elimination rules, which allow a formula containing the connective to be decomposed into its constituent parts. The rules can be succinctly expressed in the sequent calculus. In CTT there are similar introduction and elimination rules, but these rules regulate how types *and* expressions may

be manipulated. The rules for CTT are expressed in the extended sequent calculus notation given above. We have already seen examples of an introduction rule and an elimination rule above with the rules for function introduction and application. These formal rules give us a way to construct expressions in the λ-calculus and simultaneously construct the types for those expressions. As with base types, if an expression s can be constructed with type A, then s is said to be a witness or inhabitant of A.

Base types similarly have formal introduction and elimination rules. The rules for the boolean type are given in Table 1.3. Here the introduction rules state that the constants

$$\vdash \text{True} : \text{bool} \quad (\text{True-I})$$

$$\vdash \text{False} : \text{bool} \quad (\text{False-I}) \qquad \frac{\Gamma \vdash s : \text{bool} \qquad \begin{array}{c}\Delta, \text{True} : \text{bool} \vdash t : B \\ \Omega, \text{False} : \text{bool} \vdash t' : B\end{array}}{\Gamma, \Delta, \Omega \vdash \text{if}\, s\, t\, t' : B} \quad (\text{bool-E})$$

Table 1.3: CTT: rules for booleans

True and False exist and have type bool. The elimination rule states that an arbitrary boolean and two expressions of identical type, one constructed using True and the other constructed using False, give rise to an appropriate `if` expression[3]. The meaning of the `if` expression is what one would expect: the result of evaluating if $s\,t\,t'$ is the result of evaluating t if s evaluates to True, and t' if it evaluates to False. Introduction rules for the natural numbers are similar to those for the booleans, but there are infinitely many of them since there are infinitely many natural number constants. The elimination rule for the naturals presented in [27] provides a construct for primitive recursion (equivalent to *a-priori* bounded tail-recursion, or `for` loops). This is discussed further in Appendix A and Chapter 2.

Typical programming languages support structures or records and unions, functions, and function application. The λ-calculus already has functions and application; we can add special syntax for tuples and unions. Let $A_1, \ldots, A_n$ be types. Intuitively

- the type of a tuple with fields of type $A_1, \ldots, A_n$ can be represented as a conjunction $A_1 \wedge \ldots \wedge A_n$;
- the type of a union with variants $A_1, \ldots, A_n$ can be represented by a disjunction $A_1 \vee \ldots \vee A_n$; and

[3]The `if` syntax is again an extension of the λ-calculus, but can be given a definition within the regular λ-calculus.

- the type of a function from type A to type B can be represented by the implication $A \to B$.

The introduction and elimination rules for tuples, variants, and functions are given in Table 1.4. Rule names in Table 1.4 are enclosed in parentheses; "I" indicates an introduction

$$\frac{\Gamma \vdash s : A \qquad \Delta \vdash t : B}{\Gamma, \Delta \vdash (s,t) : A \wedge B}\ (\wedge\mathrm{I}) \qquad\qquad \frac{\Gamma \vdash s : A_1 \wedge A_2}{\Gamma \vdash \pi_i(s) : A_i}\ (\wedge\mathrm{E}_i)$$

$$\frac{\Gamma \vdash s : A_i}{\Gamma \vdash \kappa_i(s) : A_1 \vee A_2}\ (\vee\mathrm{I}_i) \qquad\qquad \frac{\Gamma \vdash A \vee B \qquad \begin{array}{c}\Delta, x : A \vdash t : C \\ \Omega, y : B \vdash t' : C\end{array}}{\Gamma, \Delta, \Omega \vdash \mathrm{case}_{x,y}\, s\, t\, t' : C}\ (\vee\mathrm{E})$$

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash (\lambda x.s) : A \to B}\ (\to\mathrm{I}) \qquad\qquad \frac{\Gamma \vdash s : A \to B \qquad \Delta \vdash t : A}{\Gamma, \Delta \vdash (st) : B}\ (\to\mathrm{E})$$

Table 1.4: CTT: basic rules

rule, and "E" an elimination rule. The interpretation of the rules with respect to the term language is

- $\wedge$I builds an ordered pair out of supplied components; $\wedge\mathrm{E}_i$ projects out the ith component of an existing ordered pair;

- $\vee\mathrm{I}_i$ builds variant κ_i of a disjunction, where κ_i plays the same role as the variant tag in Pascal; $\vee$E takes a term of disjunctive type and two expressions of common type which are constructed by assuming different variants. The expression $\mathrm{cases}_{x,y}\, s\, t\, t'$ is evaluated to $t[r/x]$ if $s = \kappa_1(r)$ for some r, and to $t'[r/y]$ if $s = \kappa_2(r)$ for some r;

- $\to$I takes an expression $s : B$ parameterized by an assumption $x : A$ and produces a function from type A to type B, discharging the assumption in the process; $\to$E applies a function to an argument of the correct type.

The rules given so far define a simple extension of the λ-calculus with types. Type-checking corresponds to obeying the inference rules of the logic. By applying the inference rules to build up $s : A$, we show that s has type A. Suppose that the final sequent in a derivation is $t_1 : A_1, \ldots, t_n : A_n \vdash s : A$. The structure of s records which rules were applied to build $s : A$ from the assumptions $t_1 : A_1, \ldots, t_n : A_n$. That is, the shape of s with respect to the assumed terms $t_1, \ldots, t_n$ encodes what inference rules were applied to the assumed types $A_1, \ldots, A_n$ to obtain the final type A. In fact for the rules given here, there is an

isomorphism [4] between terms and derivations: this is the Curry-Howard isomorphism. By looking at the structure of s we can reconstruct the collection of inference rules applied and the subgoals that were created (the sequent *tree*), and conversely, by knowing which inference rules were applied to obtain A from $A_1, \ldots, A_n$, we can reconstruct s itself. For this reason, the expression s is sometimes called the *proof* of A.

It can be shown that the set of rules given here satisfy some important *soundness* and consistency properties:

- if an expression can be typed, it has a unique type: if we can derive $s_1 : A_1$ and $s_2 : A_2$ following the rules and $s_1 = s_2$, then $A_1 = A_2$; and

- reduction is consistent with the type system: if we derive $s : A$ and s can be reduced to s' by some reduction rule of the λ-calculus (extended with special rules for new constructs like `if`), then $s' : A$ too.

Another important property of this particular CTT, but not of all typed λ-calculi, is that it is confluent: unlike in the untyped λ-calculus, we cannot define expressions which do not reduce to a unique normal form, and every reduction strategy is guaranteed to reach that unique normal form. Every well-typed program is guaranteed to terminate and produce a well-defined result [27]. Unfortunately, confluence rules out a number of useful combinators (special functions) that are stock features of conventional programming languages. The familiar `while`-loop is an example: we could add a combinator for it to our CTT and provide associated type rules that preserve the soundness properties, but even a correctly-typed `while`-loop will not terminate if its condition never evaluates to false.

The confluence property of simply-typable terms is an example of how type systems reduce the set of "legal" λ-calculus expressions. Type systems are used to reject some expressions as problematic or undesirable. Determining exactly what set of λ-calculus expressions are accepted by a type system is akin to determining the *completeness* properties of a set of inference rules in a logic.

As a trivial example of a derivation, consider the proof of the identity function on type A, which requires only a trivial sequent and a single application of $\rightarrow$I:

$$\frac{x : A \vdash x : A}{\vdash (\lambda x.x) : A \rightarrow A}$$

Notice that the final term $(\lambda x.x)$ encodes the use of the trivial introduction rule and the implication introduction. There may be many ways to obtain a term of type $A \rightarrow A$, but by

[4] A one-to-one and onto function.

examining the term we can read off the derivation. Two different derivations will produce different terms.

Obviously, a function of type $A \to B$ where $A \neq B$ cannot be an identity function: having the type $A \to A$ is a prerequisite if a function is purported to be the identity on A, but it is not a sufficient condition. With such a trivial example we can readily see what the function does by inspecting the term but, as in real-world programming languages, as the terms get more complicated, it gets harder to tell exactly what they do. While we might be guaranteed of writing programs that type-check and terminate, we have no guarantees that the functions accomplish some intended task. The next section introduces an enrichment of basic CTT that allows us to reason about the properties of functions we derive.

1.1.4 Constructive Specification: Predicates and Quantifiers

We now expand our CTT by adding n-ary predicate and quantified formula types, and corresponding terms. The new terms function as witnesses of properties of other terms, and the properties are stated in the types. As we shall see, this extension opens the door to a system for simultaneously specifying and creating programs.

The most fundamental predicate to add is equality. Thompson's introduction and elimination rules for equality are given in Table 1.5. There is in fact a different equality type for

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash W_{=}(s) : s =_A s}\ (=\mathrm{I}) \qquad \frac{\Gamma \vdash s : a =_A b \qquad \Delta \vdash t : C[a/x, a/y]}{\Gamma, \Delta \vdash W_s(s,t) : C[a/x, b/y]}\ (=\mathrm{E})$$

Table 1.5: CTT: rules for equality

each type that supports equality, hence the type subscript on the $=$ sign. The introduction rule states that any term is equal to itself. The elimination rule allows us to substitute one term for another in an arbitrary predicate provided the terms are equal. The terms $W_{=}(s)$ and $W_s(s,t)$ are witnesses to the equality of s with itself and to the substitution of s into t. They can be considered witnesses of a non-computational sort: they are evidence of properties of other terms, and don't really correspond to the typical terms in a real-world programming language.

Given some fundamental predicates such as equality, we can prove relationships between terms. To tie together what we can say about terms and the terms we construct, we add rules for the universal and existential quantifiers; these rules are given in Table 1.6. The introduction and elimination rules for $\forall$ look very much like those for $\to$, except that the

$$\frac{\Gamma, x : A \vdash s : P[x]}{\Gamma \vdash (\lambda x.s) : \forall x : A.P[x]}\ (\forall\mathrm{I}) \qquad \frac{\Gamma \vdash s : A \qquad \Delta \vdash t : \forall x : A.P[x]}{\Gamma, \Delta \vdash (t\,s) : P[s/x]}\ (\forall\mathrm{E})$$

$$\frac{\Gamma \vdash s : A \qquad \Delta \vdash p : P[a/x]}{\Gamma, \Delta \vdash (s, p) : \exists x : A.P[x]}\ (\exists\mathrm{I}) \qquad \frac{\Gamma \vdash s : \exists x : A.P[x]}{\Gamma \vdash \mathrm{Fst}\,(s) : A}\ (\exists\mathrm{E}_1)$$

$$\frac{\Gamma \vdash s : \exists x : A.P[x]}{\Gamma \vdash \mathrm{Snd}\,(s) : P[\mathrm{Snd}\,(s)\,/x]}\ (\exists\mathrm{E}_2)$$

Table 1.6: CTT: quantifier rules

type of the result is parameterized by the function argument. The quantified statement can make some assertion about the argument, perhaps relating it to result of some computation. Existential introduction allows us to extract a term of interest out of a predicate; the witness is a pair providing the term of interest and the witness to the property it satisfies. Within the property part, the structure of the term part is not known, and the term is referred to using a fresh name. The elimination rules allow us to project out the object of interest, or the witness of the property.

This framework of predicates and quantifiers lets us derive what we will call *(fully) specified* functions, whose types have the form

$$\forall x : A.\exists y : B.P[x, y]$$

where A is the argument type, and the result is a pair comprised of the result of the computation of interest, and a witness that this result has some relationship with the argument. For example, we can derive a specified identity function in the following way:

$$\cfrac{\cfrac{\cfrac{x : A \vdash x : A}{x : A \vdash \mathrm{W}_{=}(x) : x =_A x}}{x : A \vdash (x, \mathrm{W}_{=}(x)) : \exists y : A.x =_A y}}{\vdash (\lambda x.(x, \mathrm{W}_{=}(x))) : \forall x : A.\exists y : A.x =_A y}$$

What we have here is not quite the identity function, but an augmented function that returns both the result of the identity function and evidence that this value satisfies the required relationship with the input—that is, a proof that input and result are equal. We can use this to derive an identity function:

$$\cfrac{\cfrac{\cfrac{z : A \vdash z : A \qquad \text{the above result}}{z : A \vdash (z, \mathrm{W}_{=}(z)) : \exists y : A.z =_A y}}{z : A \vdash \mathrm{Fst}\,((z, \mathrm{W}_{=}(z))) : A}}{\vdash (\lambda z.\mathrm{Fst}\,((z, \mathrm{W}_{=}(z)))) : A \to A}$$

Notice that although this function is equivalent to the identity function, it is not as efficient. The existential type is built up and will be torn apart to get the desired result. The witness of the relationship between input and result propagates into the witness for the identity function. This is a source of *computational redundancy* in CTT, the inclusion of terms in specified functions which do not serve a useful computational purpose at runtime. The elimination of such terms is however nontrivial. It is possible that the witness of a property can itself contain an object of computational interest. For example, suppose our CTT were extended with arrays of natural numbers, and we wished to derive a sorting function. Loosely, the type of a fully-specified sorting function on (functional) arrays of naturals might be something like

$$\begin{array}{l} \forall v : \mathbf{N}\,\mathrm{array}[n]\ . \\ \quad \exists w : \mathbf{N}\,\mathrm{array}[n]\ . \\ \quad\quad \mathrm{PERM}[w, v] \wedge \\ \quad\quad\quad \forall i : \mathbf{N}\ .\ \forall j : \mathbf{N}\ .\ i \leq j \wedge j < n \rightarrow w[i] \leq w[j] \end{array}$$

where $\mathrm{PERM}[w, v]$ indicates that w is a permutation of v. Presumably, $\mathrm{PERM}[w, v]$ will be shown by deriving a permutation f on the index set $\{0, \ldots, n-1\}$ such that $\forall i : \mathbf{N}\ .\ i < n \rightarrow v[f\,i] = w[i]$. This permutation function could be recovered from the witness of the specification of the sort function, and might be computationally useful. The trick to eliminating computational redundancy from complete programs is to determine which parts of specifications are used to extract terms that will be used in computations. Techniques for reducing computational redundancy have been developed by several others and will not be discussed further in this work.

The ability to derive fully-specified functions allows us to use CTT to show that functions which solve problems of interest exist. Since the logic is constructive, functions are the witnesses of derivations, and can be extracted via the Curry-Howard isomorphism. Types can contain assertions about relationships between terms and, provided the type rules are sound, the relationships are guaranteed to hold[5]. Thus the extracted terms will be *guaranteed* to meet their specification as expressed in the types; we have a potential formal software development method that integrates specification, proof that the specification can be satisfied, and implementation.

In closing this section, we should note that the degree to which a specified function is really *fully* specified is a matter of choice and convenience. For example, return to the sorting example for the extended CTT above. We might choose to limit the type of a

[5]The type theory of this section can be shown to be sound.

specified sorting function to just

$$\begin{array}{l} \forall v : \mathbf{N}\,\mathrm{array}[n]\ . \\ \quad \exists w : \mathbf{N}\,\mathrm{array}[n]\ . \\ \quad\quad \forall i : \mathbf{N}\ .\ \forall j : \mathbf{N}\ .\ i \le j \wedge j < n \rightarrow w[i] \le w[j]. \end{array}$$

This is not a complete specification, because sorting is a permutation, and there is nothing in the specification part of the existential that indicates that w is a permutation of v. A function that returned an array twice as long and with all elements equal would also satisfy this specification. However, if the operations that have been used to derive the sorting function are all permutations themselves (for example, exchanging two elements of an array) then we might be confident that the sort function is itself a permutation. Depending on the circumstances, when we select a specification we can exercise some judgment about which properties are essential to prove and which properties are obvious and need not be proven. There are of course dangers in this approach, but it is consistent with the pragmatic attitude that formal methods should be used to reduce the risk of errors in software, rather than to eliminate it entirely.

1.1.5 The Potential for Automation

In the example derivations we have done, we could have stripped out all of the terms except for the assumptions, done the derivations in regular FOL, and then recovered the final terms by examining the rules that were applied. That we can do this is a consequence of the Curry-Howard isomorphism. For derivations on paper, doing this saves little except space. But for derivations done with the assistance of a computer, there is the potential for a kind of automatic programming.

Suppose we are given a target type for a fully-specified function, which has the form

$$\forall x : A . \exists y : B . P[x, y].$$

In principle, we might apply automated theorem proving to find a derivation of this target specification using only those rules of FOL that have counterparts in our CTT. That is, using only the constructive rules of FOL. If the computer can find such a derivation, then it can post-process it and fill in the corresponding term, which is a function that satisfies the specification embedded in its type. We have an automatic way of computing programs from their specifications.

Of course, a computer program that could do this in general is impossible, but one can imagine a system that substantially lessens the drudgery of writing programs in a CTT. We

feel that CTT provides a an attractive framework for formal programming, but one that is hampered by excessive overhead. Such a computer assistant could make programming in CTT practical. We will not consider how such a program could be built, but the possibility is a motivation for our work.

1.2 Imperative Programming

The CTT of the previous section is purely functional. There are no explicit storage cells whose values can change, or any other notion of local state. This thesis attempts to determine whether these imperative features can be added to a CTT. In this section we define "imperative" features and motivate our interest in them; in the next, we state the goals of this work explicitly.

"Imperative" originally indicated a programming model in which a program has a store or state, and runs by executing commands that would change the state. The important characteristic of imperative languages is not that the basic computational operation is a command, but that there is a state that changes as the program runs, and this state is visible to the programmer. Suppose the value associated with an identifier can be changed by a program; then we say the identifier names a *variable*[6]. The collection of variables in a program forms the explicit state that is available to the programmer. In a purely functional language such as the CTT of the last section there is no state, and a term's behavior is determined only by the arguments to it (referential transparency)[7] In contrast, in an imperative language a term's behavior may change as the program runs, either because the term is in fact a variable, or else because it makes reference to one. For example, consider the following fragment of a LISP session:

```
>(setq x 1)
1

>(defun addx (y) (+ y x))
ADDX
```

[6]Some people draw a distinction between mutable variables and immutable variables; we will simply use the term variable to refer to something whose value can change.

[7]Strictly speaking, there are no such languages in real life. Input and output are, for example non-functional features of functional languages, because the language cannot maintain a purely functional model of the outside world. For example, an input "function" will return different results on different calls. One can argue that the purpose of computation is to produce side-effects, and that the practical purpose of functional programming is to limit where and how they occur.

```
>(addx 10)
11

>(setq x 2)
2

>(addx 10)
12
```

Here the variable **x** is referred to in the body of function **addx**. Obviously, **x** is a term whose meaning (value) can change as the program runs. The example demonstrates that changing the value of **x** also changes the meaning of **addx**. Between the first invocation of **addx** and the second, the value of **x** changes and consequently so does the behavior of **addx**.

Functional programming enthusiasts sometimes criticize imperative languages as dangerous. They claim that if terms have side effects it can be easy to lose track of which terms modify the state and how, especially if there is structure sharing or other forms of aliasing. This can lead to incorrect assumptions on the part of the programmer about the values of variables and their relationship to one another, and is arguably a rich source of bugs. Personal experience suggests that much of the formal and informal effort programmers make to program carefully in imperative languages involves reasoning about keeping the state in their programs "consistent" or "legal", maintaining relationships between variables and ensuring that computations do not corrupt one another's results (interfere). Formal techniques for specifying or reasoning about imperative programs usually make the state and changes to it explicit[10].

Traditional CTT provides a way to specify purely functional programs and the potential for automatically extracting an implementation from the specification. But state forms one of the cornerstones of all procedural and most popular object-oriented languages. While these languages are in principle no more powerful than purely functional languages, the use of state often leads to more efficient programs, or to a greater degree of modularity than can be achieved with similar effort in a pure functional context [1]. Moreover, imperative languages are clearly more widely used than functional ones, and a useful formal method should be convenient to use with conventional programming languages. For these reasons, this work aims to develop a CTT for imperative programming. A complete system must

- provide a logic that has a Curry-Howard-like correspondence to an imperative term

language—that is, a term language with variables; and

- provide a means of reasoning about changes of value so that fully-specified imperative programs can be built.

The next section states our specific goals, and outlines the rest of the thesis.

1.3 Thesis Goals and Overview

The goal of this thesis is to answer the following questions:

1. Is it possible to develop a constructive type theory for a simple imperative programming language?

2. Assuming it is possible, develop a simple prototype CTT for imperative programming. Is the resulting system easy to derive programs in?

3. Could a practical tool for formal programming be built based on this CTT?

Linear logic (LL) has been suggested as a possible formal starting point for type systems for imperative languages. Chapter 2 introduces LL, which has been called a "logic of resources," and draws a distinction between resources that may be used only once, and resources that may be used any number of times. Reusable resources are given distinct names for each use. We focus on the natural-deduction style, intuitionistic fragment of linear logic (NILL). A companion logic for NILL that allows repeated use of the names of reusable resources is developed, and the extension is shown to add no expressive power.

Chapter 3 extends the logic developed in Chapter 2 to a constructive type theory for a λ-calculus with variables and assignment. We begin by describing an extension of NILL developed by Reddy that makes it more convenient for typing an imperative language based on Reynolds' syntactic control of interference. We then extend Reddy's version of linear logic to a full CTT for the λ-calculus with variables, arrays, and assignment in the same spirit as the CTT presented in the previous sections. In this CTT, called ICTT, the features of linear logic are exploited to keep track of which objects interfere with one another. This record of interference allows types to represent statements about the contents of mutable objects.

The thesis is intended to be exploratory, to uncover promising techniques for building a constructive type theory for imperative programming, and to determine whether such a type theory is likely to be useful. While soundness issues are not ignored and we try to

lay a reasonable foundation for exploring them, a formal soundness proof for ICTT is not attempted. In fact, later chapters show that ICTT is not sound and assertions about the contents of mutable storage may be incorrect. Despite this, we believe that the informal arguments of Chapter 3 strongly suggest that a provably sound reformulation of the rules can be found.

Chapters 4 through 6 evaluate whether ICTT has any practical application, and are shorter and easier to read than Chapters 2 and 3. Chapter 4 presents an implementation a proof-checker for ICTT, and Chapter 5 describes a partial derivation of a bubble-sort function. ICTT proves awkward to work with, and derivations for even simple programs are more lengthy than those for corresponding functional programs. We also discover soundness problems. Chapter 6 concludes with an assessment of ICTT and answers questions 2 and 3 above.

1.4 Previous Related Work

This section briefly describes related work. Most of this work has focused on developing type systems for imperative λ-calculi some of which is discussed using terminology that will not be explained until later chapters.

There has been considerable interest in the functional programming language world in finding ways to integrate computations with state into pure functional frameworks. See for examples Hudak on mutable abstract data types [11], Odersky *et. al.* [17] on the typed lambda calculus with assignment, Garrigue and Ait-Kaci on the transformation calculus [6][7][8] , Benton and Wadler on the relationship between linear logic and monads [5], and Wadler [30][31] on monadic approaches to state.

In his Ph.D. thesis, Swarup devised a type system for imperative languages that was stratified into four layers, developing similar but separate sets of inference rules for applicative and mutable types and providing additional layers of the type system for reasoning about effects and interference [25][26][24]. This work was intended as a type system for an imperative language, but could have been used to develop a constructive system. The logic would have been quite large and complex however, and an implementation of it as a CTT was not central to the thesis and was hence not attempted.

Linear logic was developed by Girard [9] and a corresponding abstract machine was described by Lafont [14]. LL's use as a type system for functional languages with no garbage collection or simplified garbage collection has been described by Lafont[14] and Mackie [16]. There are several gentle introductions to linear logic listed in the bibliography.

The main reference for linear logic in this thesis is Troelstra [28], and the notation used here is based on his[8].

Wadler has described the interpretation of linear types as types permitting destructive update [33][32]. The basic idea is that if a function has type $A \multimap A$, then since the argument is immediately disposed of it can be overwritten with the result of the call. Thus the allocation process normally required in an applicative language is short-circuited. But the approach doesn't address the problem of variables, named objects that are accessible to the programmer and subject to destructive update.

Possible extensions of linear logic with modalities that describe mutable objects were originally investigated by Reddy [20][21], again in the context of developing type systems for imperative programming languages. He added a special *regenerative type* modality and then used it to develop a modality for describing a objects that change value but retain the same name. This work exploits Reddy's ideas by similarly developing a modal extension of NILL, but also adds the necessary features to reason about sequences of updates and develop a system for constructive specification.

[8]Unfortunately, Troelstra's notation is significantly different from Girard's, and appears to be the less widely used. However, his selection of connective symbols has a symmetry that makes their meanings easier to remember

Chapter 2

Foundations: A Linear Constructive Type Theory

In this chapter we present a type theory similar to that in Chapter 1, but based on a natural-deduction-style intuitionistic fragment of Girard's linear logic (LL) [9][28]. We call the resulting type theory linear constructive type theory (LCTT), and will use it as a basis for the further developments in Chapter 3. LCTT will include equality and quantifiers and be endowed with boolean and natural number base types, the latter permitting primitive recursion. As the type theory is developed, we will discuss the intended layout of terms in memory in preparation for Chapter 3. Finally, we will describe a modified form of LCTT that forsakes strict linearity but is easier to develop programs in.

2.1 Linear Logic Basics

Linear logic (LL) has been called a *resource-conscious* logic [9][29], because formulas are treated as resources which are *consumed* in the course of reasoning. To understand this statement, we must describe the essential differences between LL and classical first-order logic (FOL).

In FOL, once a proposition is proven it is simply considered a fact that can be reused as much as required, or discarded if it is not needed. The notions of duplication and discarding can be expressed by the *contraction rule* and the *weakening rule*. A typical sequent calculus form of these rules for a natural deduction system is

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}(\mathrm{C}) \qquad\qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash B}(\mathrm{W}).$$

The weakening rule says that we can dispose of a conclusion if we don't need it, or that we can introduce unnecessary assumptions into a derivation. Working forward, the contraction rule says that two copies of a proposition may be collapsed into one; working backwards

from the goal it says that a proposition may be freely copied. Because these rules are taken for granted in FOL, they are not normally stated explicitly.

In LL, the contraction and weakening rules are instead explicit, and their use is restricted to the so-called *banged* propositions, propositions which have a special modality indicated by a bang(!). There are also rules for forming a banged proposition from other banged propositions (promotion or !-introduction), and for demoting a banged proposition to an unbanged one. One of the consequences of this explicit control over contraction and weakening is that the meanings of the remaining logical connectives must be modified: LL has two types of conjunction and disjunction, known as the *multiplicative* and *additive* types. The complete set of LL operators and constants is

$\top$	additive unit ("true")
$\bot$	additive co-unit ("false")
$\mathbf{1}$	multiplicative unit ("one" or "unit")
$\mathbf{0}$	multiplicative co-unit ("zero")
$\star$	multiplicative conjunction ("tensor")
$+$	multiplicative disjunction ("parallel or")
$\sqcap$	additive conjunction ("and" or "your-choice")
$\sqcup$	additive disjunction ("or" or "their-choice")
$\multimap$	(multiplicative) linear implication
$\sim$	linear negation
$!$	storage modality ("bang" or "of course")
$?$	co-storage modality ("why not")
$\exists$	existential operator
$\forall$	universal operator

Most of the operators and constants in LL fall into a neat collection of DeMorgan dualities[1], given in Table 2.1. As well, we have theorems $\vdash (A \multimap B) \multimap (\sim A + B)$ and $\vdash \sim\sim A \multimap A$,

$\mathbf{1}$	$\mathbf{0}$
$\top$	$\bot$
$\star$	$+$
$\sqcap$	$\sqcup$
$\sim$	$\sim$
$!$	$?$
$\forall$	$\exists$

Table 2.1: DeMorgan Duals

which correspond to the familiar $\vdash (A \rightarrow B) \rightarrow (\neg A \vee B)$ and $\vdash \neg\neg A \rightarrow A$ from classical FOL.

[1]Two operators $F(x_1, \ldots, x_n)$ and $G(x_1, \ldots, x_n)$ are (DeMorgan) duals if $\sim F(x_1, \ldots, x_n)$ is equivalent to $G(\sim x_1, \ldots, \sim x_n)$ for all choices of $x_1, \ldots, x_n$[28].

We can roughly describe the meanings of the various connectives from the resource-consciousness perspective. The multiplicative conjunction $A \star B$ means that both A and B are available given the current resources, because they depend on different resources. In contrast, the additive conjunction $A \sqcap B$ means that while both A and B are *potentially* available given the current resources, only one of them can be obtained. That is, A and B are mutually exclusive, because they each require the same resources to build. The additive disjunction $A \sqcup B$ means that both A and B are potentially available with current resources, but only one of them will be, and we do not know which one in advance. The multiplicative disjunction $A + B$ is somewhat more subtle; it means that we can assume that at least one of A and B can be made available with current resources (this interpretation can be obtained using the DeMorgan dual identity with $\star$ and the theorem $\vdash\ \sim\sim A \multimap A$). The linear implication $A \multimap B$ indicates that B can be produced by consuming A. The negation $\sim A$ indicates that A is not available. The storage or "of course" type $!A$ means that we have an inexhaustible supply of A, and the costorage or "why not" type $?A$ means that A is not indefinitely unavailable (we do not have an infinite supply of $\sim A$) [28].

Full LL is not necessary for our purposes; we will use only a natural-deduction-style intuitionistic fragment of it, which discards multiplicative disjunction $+$, negation $\sim$, the costorage modality $?$, and the constant $\mathbf{0}$. The formal inference rules for this fragment are presented in the next section; the complete set of LL inference rules appears in the appendices.

To illustrate the use of the connectives and give the reader some intuition about them, we include a paraphrased example that is based on one in [29]. Suppose we walk into a diner to buy lunch, and a (rather limited) menu on a hanging blackboard reads as follows. We can have our choice of soup or salad with either a baked potato or french fries, and our choice of soft drink. The soup will either be cream of broccoli or minestrone; the chef decides which to make on any given day. Soft drink choices include Coke and root beer. We have five dollars in our pocket, and the total price of the lunch special is four dollars. Let the various foods and currencies be

B cream of broccoli soup

M minestrone soup

S salad

P baked potato

F french fries

C Coke

R root beer

D a dollar

Then we can describe the content of the menu as

$$(D \star D \star D \star D) \multimap ((B \sqcup M) \sqcap S) \star (P \sqcap F) \star (C \sqcap R)$$

or, equivalently

$$(D \star D \star D \star D) \vdash ((B \sqcup M) \sqcap S) \star (P \sqcap F) \star (C \sqcap R).$$

The conjunction means that each of

- our choice of either the soup the chef chose to make or the salad;
- our choice of either a potato or french fries; and
- our choice of either coke or root beer

are available. Letting LUNCH $= ((B \sqcup M) \sqcap S) \star (P \sqcap F) \star (C \sqcap R)$ we can derive

$$\dfrac{\dfrac{D \star D \star D \star D \vdash \text{LUNCH} \qquad D \vdash D}{D, D \star D \star D \star D \vdash \text{LUNCH} \star D} \qquad D \star D \star D \star D \star D \vdash D \star D \star D \star D \star D}{D \star D \star D \star D \star D \vdash \text{LUNCH} \star D}$$

or, equivalently

$$(D \star D \star D \star D \star D) \multimap ((B \sqcup M) \sqcap S) \star (P \sqcap F) \star (C \sqcap R) \star D.$$

That is, if we have five dollars then we can buy lunch and still have one dollar left over.

LL allows us to express notions of consumption, collections of items that must be all consumed together, and various kinds of alternatives in a natural way. The bang allows us to express inexhaustible resources: if we had $!D$ rather than the five dollars in our pockets, we would have a never-ending supply of dollars, and using a combination of the dereliction and contraction rules we could show that $\vdash\ !D \multimap D \star D \star D \star D \star !D$ and so

$$!D \multimap ((B \sqcup M) \sqcap S) \star (P \sqcap F) \star (C \sqcap R) \star !D$$

To conclude this section, we note that the elimination rules for ! can be replaced with the rule

$$\vdash\ !A \multimap \mathbf{1} \sqcap A \sqcap (!A \star !A)$$

so that given $!A$, we have our choice of discarding it, demoting it to A, or duplicating it and using both copies.

2.2 LCTT Inference Rules and Terms

The fragment of LL that we will concentrate on is NILL, a natural-deduction style, intuitionistic linear logic. We will parallel the development of CTT presented in Chapter 1 by first discussing NILL without equality or quantifiers and presenting a corresponding term language. We will also discuss a possible layout of terms in memory that distinguishes banged and unbanged types in the implementation of the term language. We will then add equality (and discuss predicates in general) and quantifiers, and boolean and natural number base types.

2.2.1 LCTT Without Equality or Quantifiers

A sequent calculus version of the inference rules for NILL without equality or quantifiers appears in Table 2.2. Introduction rules are on the left, and elimination rules on the right. We include corresponding terms for the formulas which are based on term assignments appearing in [28] and [29]. A term and its type are separated by a colon or a bold colon (**:**). Usually, terms which are parameters (solitary identifiers such as x) are separated from their types with a regular colon, and more complex constructions with a bold colon if it improves readability. We will call a term together with its corresponding type a typed term, or TT. Briefly, the rules are

AI: assumption introduction; a sequent of the form $x : A \vdash x : A$ is called a *trivial* sequent.

⊤I: additive unit introduction. From any collection of assumptions we can conclude a TT Voida $: \top$, where Voida is the name of the unique witness for $\top$. $\top$ has the property that

$$\top \sqcap A \equiv A \sqcap \top \equiv A$$

⊥E: $\perp$ plays the role of F in classical logic, and this rule is the linear logic equivalent of "from a false statement one can prove anything." The term Abort$_x$ can be declared to have any type; its interpretation in the term language is that the program aborts the computation because it has encountered a problem witnessed by x.

1I: multiplicative unit introduction. Without any assumptions we can conclude a TT Voidm $: \mathbf{1}$, where Voidm is the name of the unique witness for $\mathbf{1}$. $\mathbf{1}$ has the property that

$$\mathbf{1} \star A \equiv A \star \mathbf{1} \equiv A$$

$$x:A \vdash x:A \quad (\mathrm{AI})$$

$$\Gamma \vdash \mathrm{Voida} : \top \quad (\top\mathrm{I}) \qquad \Gamma, x:\bot \vdash \mathrm{Abort}(x) : A \quad (\bot\mathrm{E})$$

$$\vdash \mathrm{Voidm} : \mathbf{1} \quad (\mathbf{1}\mathrm{I}) \qquad \frac{\Gamma \vdash s:\mathbf{1} \qquad \Delta \vdash t:A}{\Gamma, \Delta \vdash s;\ t : A} \ (\mathbf{1}\mathrm{E})$$

$$\frac{\Gamma \vdash s:A \qquad \Delta \vdash t:B}{\Gamma, \Delta \vdash (s,t) : A \star B} \ (\star\mathrm{I}) \qquad \frac{\Gamma \vdash s : A \star B \qquad \Delta, x:A, y:B \vdash t:C}{\Gamma, \Delta \vdash \mathrm{let}\ (x,y) = s \ \mathrm{in}\ t : C} \ (\star\mathrm{E})$$

$$\frac{\Gamma \vdash s:A \qquad \Gamma \vdash t:B}{\Gamma \vdash \langle s,t \rangle : A \sqcap B} \ (\sqcap\mathrm{I}) \qquad \frac{\Gamma \vdash s : A_0 \sqcap A_1}{\Gamma \vdash \pi_i s : A_i} \ (\sqcap\mathrm{E}_i)$$

$$\frac{\Gamma \vdash s:A_i}{\Gamma \vdash \kappa_i\, s : A_0 \sqcup A_1} \ (\sqcup\mathrm{I}_i) \qquad \frac{\Delta \vdash s : A \sqcup B \qquad \begin{array}{c}\Gamma, x:A \vdash t:C \\ \Gamma, y:B \vdash t':C\end{array}}{\Gamma, \Delta \vdash \mathrm{case}\ s\ t\ t' : C} \ (\sqcup\mathrm{E})$$

$$\frac{\Gamma, x:A \vdash t:B}{\Gamma \vdash \lambda x.t : A \multimap B} \ (\multimap\mathrm{I}) \qquad \frac{\Gamma \vdash s : A \multimap B \qquad \Delta \vdash t:A}{\Gamma, \Delta \vdash s\,t : B} \ (\multimap\mathrm{E})$$

$$\frac{!\Gamma \vdash t:A}{!\Gamma \vdash \mathrm{Store}\,(t) : !A} \ (!\mathrm{I}) \qquad \frac{\Gamma \vdash s: !B \qquad \Delta, x:B \vdash t:A}{\Gamma, \Delta \vdash \mathrm{let}\ x = \mathrm{Get}\,(s)\ \mathrm{in}\ t : A} \ (!\mathrm{E})$$

$$\frac{\Gamma \vdash s: !B \qquad \Delta \vdash t:A}{\Gamma, \Delta \vdash \mathrm{Ignore}\,(s)\,;\ t : A} \ (!\mathrm{Ew})$$

$$\frac{\Gamma \vdash s: !B \qquad \Delta, x: !B, y: !B \vdash t:A}{\Gamma, \Delta \vdash \mathrm{let}\ x = y = s\ \mathrm{in}\ t : A} \ (!\mathrm{Ec})$$

Table 2.2: LCTT basic rules

1E: multiplicative unit elimination. $s;t$ represents a sequence of two computations: first computation s is performed and the (otherwise useless) $\mathbf{1}$ type result is discarded; then computation t is performed. The type of the sequence is the type of t.

$\star$I: multiplicative conjunction introduction. Given two TTs $s : A$ and $t : B$ derived from different assumptions, we can derive the *strict* pair $(s,t) : A \star B$. In order to preserve linearity, both parts of the pair must be used in any computation that consumes the pair. The pair is called strict because both components will be used, and hence it makes sense to evaluate them before building the pair.

$\star$E: multiplicative conjunction elimination. Given a strict pair $s : A \star B$ and a term $t : C$ derived from different assumptions including $x : A$ and $y : B$, form a term by substituting the components of the pair for x and y. The expression on the left of the equals sign in

$$\text{let } (x, y) = s \text{ in } t$$

is a dismantler or destructor pattern. It means that s has the structure of a strict pair, and it is to be evaluated and dismantled, its components put into x and y. In the CTT of Chapter 1, we simply used the Fst and Snd special functions, but with the strict pair of linear logic we must ensure that *both* of the components of s are extracted and used in a computation.

$\sqcap$I: additive conjunction introduction. Given $s : A$ and $t : B$ derived from exactly the same assumptions, form the *lazy* pair $\langle s, t\rangle : A \sqcap B$. In order to preserve linearity, only one of the disjuncts can be used in a computation that consumes the pair. The pair is called lazy because none of the components need be computed until the desired component is projected out (it is a waste of computational effort to evaluate the components and then build the pair, because one of them will be discarded).

$\sqcap$E$_i$: additive conjunction elimination. Project out the ith component of a lazy pair, discarding the rest. Notice that $\star$ and $\sqcap$ together have the character of traditional conjunction.

$\sqcup$I$_i$: additive disjunction introduction. Form one of the possible variants of a disjunctive type. This corresponds to the disjunctive type in CTT.

$\sqcup$E: additive disjunction elimination. Given a term $s : A \sqcup B$ and terms t and t' which are

- of the same type C;

- are derived from the same sets of assumptions as one another, except for assuming different possible variants of the disjunction; and
- are derived from different assumptions than the disjunction

we can derive a TT case $s\,t\,t' : C$ whose value is $t[s/x]$ if s is of first-variant type, or $t'[s/x]$ if s is of second-variant type. The assumptions Γ can be common to the derivations of $t:C$ and $t':C$ because only one of them will actually be used.

$\multimap$ I: (linear) implication introduction. If we assume $x : A$ in deriving $t : B$, then we can abstract $x : A$ away and form the function TT $(\lambda x.s) : (\forall x : A.B)$.

$\multimap$ E: (linear) implication elimination. This is function application. Given $s : A \multimap B$ and $t : A$ derived from different assumptions we can derive a TT $(s\,t) : B$. We interpret linear function application as consuming the argument to produce the result.

!I: bang introduction. Any TT derived from only banged assumptions can be banged (and hence used repeatedly, albeit under different names). The expression $\mathrm{Store}\,(t)$ can be loosely interpreted as "put t in storage so we can remember it and make copies of it."

!E: bang elimination (or *dereliction*). Given $s : !B$ and $t : A$ derived from different assumptions including $x : !B$, we can derive $t[\mathrm{Get}\,(s)\,/x] : A$. In other words, a copy of s is fetched and used in a computation.

!Ew: weakening. Given $s : !B$ and any independently-derived $t : A$ we can ignore s and carry on with t. Terms of unbanged type, except those terms of multiplicative unit ($\mathbf{1}$) type, cannot be ignored in this way.

!Ec: contraction. Given $s : !B$ and $t : A$ derived from different assumptions including $x : !B$ and $y : !B$, we can derive a term equivalent to $t[s/x, s/y] : A$. That is, we can copy s into x and y.

Like the basic CTT of Chapter 1, these rules together with some base types give us a strongly typed functional programming language, where correct typing corresponds to obeying the inference rules. Programs in this language have the interesting property that in all function and **let** bodies each symbol appears exactly once [28]. This is the manifestation of linearity within the term language. We can propose a memory layout for terms and hence give an interpretation of banged and unbanged types with respect to the implementation of the linear term language. Every identifier represents a single-use pointer to a memory cell.

By single use we mean that once the symbol is used in a computation (*i.e.* its type has been used as an argument to an inference rule) it is no longer available for use. The storage cells corresponding to terms of banged types are special, however; they contain pointers to other storage cells, and these pointers can be dereferenced (dereliction), copied into other pointer cells (contraction), or simply discarded (weakening). From the perspective of storage management, the contents of a storage cell of type A can be used multiple times if and only if it is the target of one or more pointer cells of type $!A$. If there are no such pointers for an object, its storage can be reclaimed as soon as the object is used; otherwise, it must be garbage collected. Figure 2.1 illustrates this storage interpretation pictorially. Ordinary storage cells are indicated by empty boxes, and the special pointer cells are indicated by blackened boxes. A storage cell can correspond to the result of some aggregate type such as a strict tuple, and so can contain other storage cells. The figure shows the effect of the three bang elimination rules given this interpretation of terms

contraction elimination !Ec takes an existing $!A$ pointer and a structure parameterized by two unassigned pointers of type $!A$, and copies the pointer into the two parameters. The original pointer is discarded.

dereliction or plain !E takes an existing $!A$ pointer and a structure parameterized by an unassigned object of type A and sets the undefined object to the pointed-to object. The original $!A$ pointer is discarded.

weakening !Ew takes an existing $!A$ pointer and discards it; an unrelated structure can be the result of the computation. The pointed-to object may be discarded if the discarded pointer was the last pointer pointing to it.

This is not the only possible interpretation of the types with respect to memory layout. Some research has been done on exploiting linear typing in functional languages to simplify (and perhaps speed up) automatic storage allocation by eliminating complicated garbage collection [14]. Eliminating garbage collection requires that all data structures be trees, so the copy operation corresponding to contraction must be a deep copy, rather than a simple pointer copy, to avoid sharing. That is, every Get(s) is a deep copy of s, rather than a copied pointer to s. This entails an obvious space penalty, and we will subscribe to the copied-pointers interpretation in this work. Baker[4] has proposed a scheme with limited pointer sharing that uses hash consing and simplified reference-count garbage collection as a compromise on the deep-copy technique that mantains many of its simple storage allocation properties but increases its efficiency.

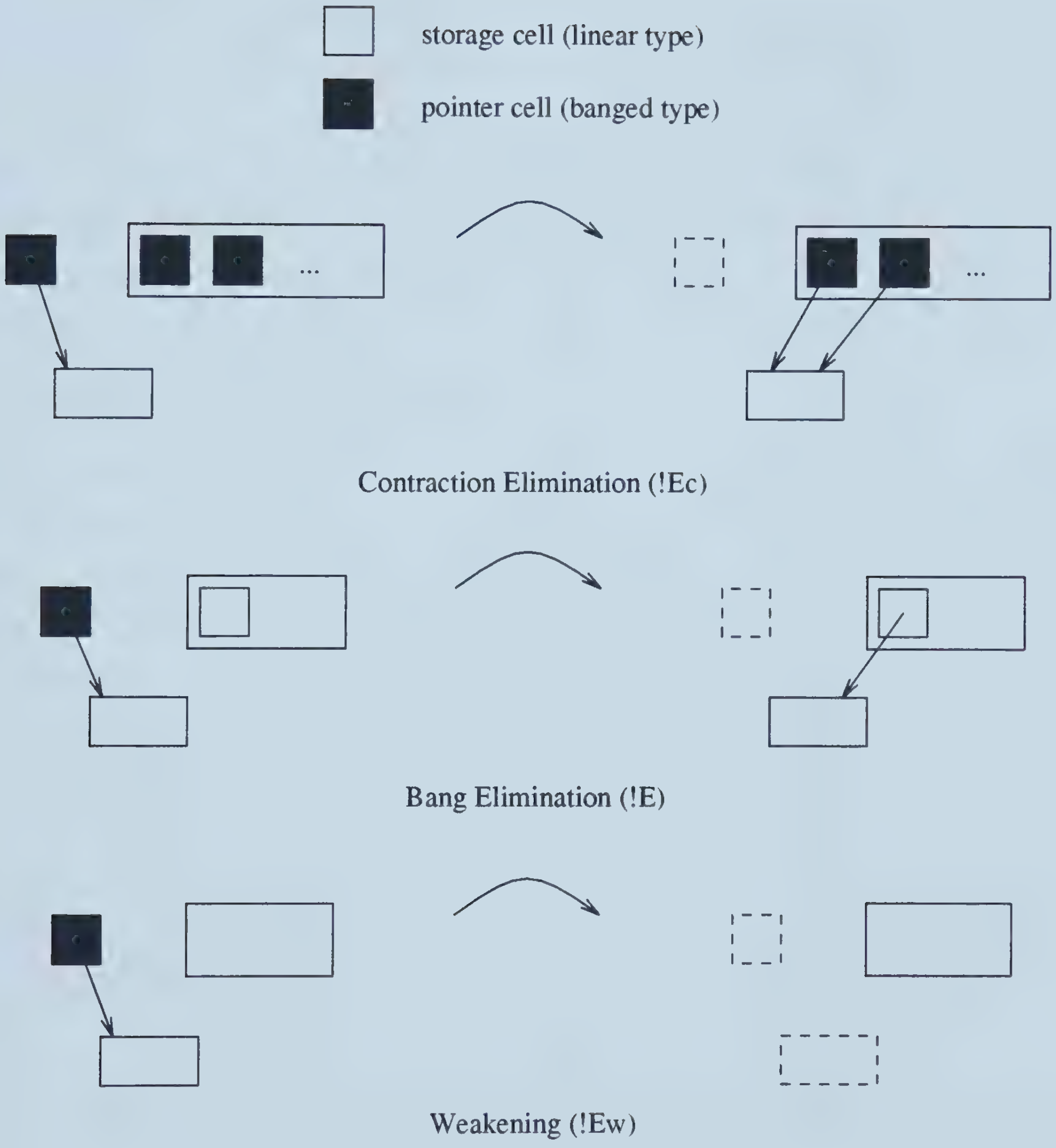
storage cell (linear type)
pointer cell (banged type)
...
...
Contraction Elimination (!Ec)
Bang Elimination (!E)
Weakening (!Ew)

Figure 2.1: Banged types as pointers that can be copied

Wadler [29] has suggested that terms with linear type can be interpreted as corresponding to objects that permit destructive update. For example, let A be some linear type and B some other type, and suppose we can build a function of type

$$A \multimap B \multimap A.$$

Then since the first argument is of linear type, it can be considered discarded once fed into the function, and the compiler could arrange for the function result to overwrite the storage occupied by the argument. The relationship to imperative programming is however somewhat indirect; in a true linear language, identifiers are not reused, and so though the argument and result may occupy the same storage and the result can be viewed as an updated version of the argument, it is given a different name. Within the language itself, everything still appears applicative.

2.2.2 Predicates and Quantifiers

In order to turn LCTT into a system within which we can write and reason about programs, we need to allow predicates and quantified formulas as in CTT. In particular we will need equality. Recall the CTT equality rules given in Table 1.5. We could mimic those rules exactly, and add the null-intersection constraints on left sequents as in the other LCTT rules, but equality will serve our purposes better if we modify the rules slightly. The equality introduction rule has a new form with an empty left side in the resulting sequent. This is because an equality assertion does not consume the terms it pertains to; having the terms appear in the left context would disallow computing with an object and simultaneously reasoning about it. We regard $x =_A x$ as a truth about the term $x : A$ rather than something computed from it. Thus we only require that x involves existing resources and that we can verify it has type A. The revised equality rules appear in Table 2.3.

$$\vdash W_=(s) : s =_A s \quad (=\mathrm{I}) \qquad \frac{\Gamma \vdash s : a =_A b \qquad \Delta \vdash t : C[a/x, a/y]}{\Gamma, \Delta \vdash W_s(s, t) : C[a/x, b/y]} \quad (=\mathrm{E})$$

Table 2.3: LCTT: rules for equality

We support equality only for *passive discrete types*[2]. Intuitively, a datum with discrete type represents a solitary, well-defined object like an integer or boolean value. The $\star$ and

[2]Discrete types are discussed in Appendix B; they include simple base types like natural numbers and booleans and are closed under constructions using $\star$, $\sqcup$ and the existential quantifier. Under certain limited circumstances function and universally-quantified types can be discrete. The passive discrete types exclude these.

$\sqcup$ connectives preserve discreteness, so a pair of discrete objects and a discrete object with a variant tag are themselves discrete objects. A datum of non-discrete type represents a collection of choices or a suspended or lazy computation. The $\sqcap$, $\multimap$ (usually) and ! types are all examples of non-discrete types. The *passive* discrete types exclude the few discrete function types. Thus we permit equality on traditional *data* types, but not on functions and lazy constructions like the non-strict pair $\langle s, t\rangle$. The issue of discrete *versus* non-discrete types will appear again in the development of imperative features in Chapter 3.

The quantifier rules of LCTT appear in Table 2.4. We have

∀I: the same as ∀I in CTT;

∀E: the same as in CTT, except Γ and Δ are disjoint;

∃I: the same as in CTT, except Γ and Δ are disjoint. The existential pair is strict.

∃E: here because the existential pair is strict[3] we cannot have projections; both components must be consumed at once as in $\star$E. Again, Γ and Δ are disjoint.

Although the syntax of the quantified formulas is identical to that for CTT, and the inference rules are similar, the interpretation of the formulas is fundamentally different. The existential is interpreted as

$$\bigsqcup_{x \in A} (x, P[x]),$$

a kind of giant additive disjunction or "their-choice." That the object of interest can be renamed in the ∃I rule indicates anonymity: $\exists x : A.\, P[x]$ could represent *any* element $x \in A$ together with a corresponding $a \in P[x]$. The eliminating term in ∃E cannot make any assumptions about the identity of x, just as the $\sqcup$E rule must handle all possible variants of the disjunction. The universal is correspondingly interpreted as

$$\bigsqcap_{x \in A} (x, P[x]),$$

a kind of giant additive conjunction or "your-choice." The type $\forall x : A.\, P[x]$ presents a choice of $P[x]$ for any $x \in A$; but once we have made the choice through ∀E, the other alternatives become unavailable. This is not the same as the interpretation of the universal in CTT: since it can be freely reused, $\forall x : A.\, P[x]$ implies $P[x]$ for every $x \in A$ simultaneously. The corresponding formula in linear logic is $!\forall x : A.\, P[x]$. We will shortly provide a rigorous justification for this interpretation of the quantifier rules.

[3]The brackets for the existential pair and the lambda for the universal are boldface to distinguish them from the regular strict pair brackets and lambda.

$$\frac{\Gamma, x : A \vdash t : P[x]}{\Gamma \vdash (\boldsymbol{\lambda} x.t) : \forall x : A.\, P[x]} (\forall \mathrm{I}) \qquad \frac{\Gamma \vdash s : A \qquad \Delta \vdash t : \forall x : A.\, P[x]}{\Gamma, \Delta \vdash (t\, s) : P[s/x]} (\forall \mathrm{E})$$

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash s : P[t]}{\Gamma \vdash (t, P[t]) : \exists x : A.\, P[x]} (\exists \mathrm{I})$$

$$\frac{\Gamma \vdash s : \exists y : A.\, P[y] \qquad \Delta, x : A, w : P[x] \vdash t : B}{\Gamma, \Delta \vdash \mathrm{let}\ (x, w) = s\ \mathrm{in}\ t : B} (\exists \mathrm{E})$$

x is not free in Γ in $\forall$I and not free in Δ or B in $\exists$E

Table 2.4: LCTT: quantifier rules

These rules are also a little different than what is suggested by the corresponding rules for NILL without the term language given in [28], because in NILL there is no association

$$\frac{\Gamma \vdash P[y/x]}{\Gamma \vdash \forall x.P} (\forall \mathrm{I}) \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[t/x]} (\forall \mathrm{E})$$

$$\frac{\Gamma \vdash P[t/x]}{\Gamma \vdash \exists x.P} (\exists \mathrm{I}) \qquad \frac{\Gamma \vdash \exists x.P}{\Delta, A[y/x] \vdash C} (\exists \mathrm{E})$$

Table 2.5: Original NILL quantifier rules

between terms and propositions. Terms are simply symbolic arguments to quantifiers, or the quantified objects in quantified formulas. But in LCTT every term has an associated formula and hence an associated left sequent. Terms are the resources that the propositions describe and the left contexts of terms must be taken into account to preserve linearity. Thus $\forall$E looks more like $\multimap$E now and there is an analogous linearity constraint, and $\exists$E makes both the quantified object and the witness of its property available in a way consistent with a strict pair —the sort of term produced by the introduction rule dictates how the term can be taken apart in the elimination rule.

Coherence Space Semantics of Predicates and Quantifiers

The treatment of predicates and quantifiers presented in this section can be justified in the coherence space semantics for linear logic. Coherence spaces and the coherence space semantics of LL are presented in Appendix B. The material here can be skipped on a first reading.

Let $P[x_1 : A_1, \ldots, x_n : A_n]$ be a base predicate. Then with each instantiation $P[s_1, \ldots, s_n]$ of P for elements $s_1 \in A_1, \ldots, s_n \in A_n$ we associate a distinct type whose web is

$$\mathcal{A}_{P[s_1,\ldots,s_n]} := (P, s_1, \ldots, s_n)$$

and whose coherence relation is

$$a \sim_{P[s_1,\ldots,s_n]} a' \iff a = a'.$$

The type $P[s_1, \ldots, s_n]$ is thus discrete and single-atom, with elements $\mathrm{Abort}_{P[s_1,\ldots,s_n]} := \emptyset$ and $\{(P, s_1, \ldots, s_n)\}$, which can be interpreted as indicating that $P[s_1, \ldots, s_n]$ is false or true, respectively. We include the predicate name in the single atom so that the webs for P are distinct from the webs of any other predicate with the same arity.

To an existential type $\exists x : A.\, P[x]$ we assign the web

$$\mathcal{A}_{\exists x:A.\, P[x]} := \overset{\circ}{\bigcup_{x \in A}} \{x\} \times \mathcal{A}_{P[x]}$$

where $\overset{\circ}{\bigcup}$ is disjoint union, and coherence relation

$$(x, a) \sim_{\exists x:A.\, P[x]} (y, b) \iff x = y \;\wedge\; a \sim_{P[x]} b.$$

This web and coherence relation are similar to that for the additive disjunction, except that the index set is a type. This is consistent with the statement that the existential type is a giant "somebody else's choice." The elements of $\exists x : A.\, P[x]$ are thus pairs, each with some element $x \in A$ as the first component, and an element $W \in P[x]$ as the second. Note that P need not be a base predicate, but any formula in which x can be substituted, the type $P[x]$ need not be discrete. In the event that it is, then $\sim \exists$ is equality and $\exists x : A.\, P[x]$ is a discrete type.

To a universal type $\forall x : A.\, P[x]$ we also assign the web

$$\mathcal{A}_{\forall x:A.\, P[x]} := \overset{\circ}{\bigcup_{x \in A}} \{x\} \times \mathcal{A}_{P[x]}$$

but the coherence relation is

$$(x, a) \sim_{\forall x:A.\, P[x]} (y, b) \iff x \neq y \;\vee\; (x = y \wedge a \sim_{P[x]} b).$$

This choice of web and coherence relation is consistent with the statement that a universal is like a "giant your-choice." Note that, like the function type $A \multimap B$, $\forall x : A.\, P[x]$ is discrete

if and only if $P[x]$ is discrete and A is single-atom. The elements of $\forall x : A. P[x]$ are sets of the form

$$\overset{\circ}{\bigcup_{x \in A'}} \{x\} \times \beta_x \quad \text{where } A' \subseteq A \text{ is arbitrary and } \beta_x \in P[x].$$

Since $\sim_{P[x]}$ is a coherence relation, it is reflexive and symmetric and thus obviously so are both $\sim_{\exists x:A.\,P[x]}$ and $\sim_{\forall x:A.\,P[x]}$, so the relations are indeed valid coherence relations. Having made these definitions, we must be able to use them to justify the inference rules stated for existential and universal types.

Lemma 1 For each $x \in A$ there is a linear function $f_{\exists \mathrm{I}} : P[x] \multimap \exists y : A.\,P[y]$ such that $f_{\exists \mathrm{I}}(s) = \{x\} \times s$ for every $s \in P[x]$.

Proof. To define a linear function we must first select a pairing of atoms in $P[x]$ with atoms in $\exists y : A.\,P[y]$ and then show that the pairing respects the coherence relation for $P[x] \multimap \exists x : A.\,P[x]$. We pair exactly those $(\alpha) \in \mathcal{A}_{P[x]}$ and $(y, \beta) \in \mathcal{A}_{\exists y:A.\,P[y]}$ for which $x = y$ and $\alpha = \beta$ and verify that:

$$\begin{array}{l}(\alpha, (x, \alpha)) \sim_{P[x] \multimap \exists y:A.\,P[y]} (\alpha', (x', \alpha')) \\ \Longleftrightarrow \\ \alpha \sim_{P[x]} \alpha' \Rightarrow ((x, \alpha) \sim_{\exists y:A.\,P[y]} (x', \alpha') \wedge ((x, \alpha) = (x', \alpha') \Rightarrow \alpha = \alpha'))\end{array}$$

From the definition of the pairing and $\alpha, \alpha' \in P[x]$ we must have $x = x'$. Assuming that $\alpha \sim_{P[x]} \alpha'$ we then require

$$\begin{array}{c}(x, \alpha) \sim_{\exists y:A.\,P[y]} (x', \alpha') \\ \Longleftrightarrow \\ x = x' \ \wedge \ \alpha \sim_{P[x]} \alpha'\end{array}$$

which is easily shown given the above remarks. That $(x, \alpha) = (x', \alpha') \Rightarrow \alpha = \alpha'$ is obviously true. Thus this pairing does indeed define a linear function; call it f.

Now let $\alpha \in s \subseteq \mathcal{A}_{P[x]}$. Then $(\alpha, (x, \alpha)) \in \mathcal{A}_{P[x] \multimap \exists y:A.\,P[y]}$ is the only pairing involving α. Thus $f(\{\alpha\}) = \{(x, \alpha)\}$. But since f is linear it commutes with arbitrary unions, so

$$f(\alpha) = f\Big(\bigcup_{\alpha \in s} \{\alpha\}\Big) = \bigcup_{\alpha \in s} f(\{\alpha\}) = \bigcup_{\alpha \in s} \{(x, \alpha)\} = \{x\} \times s$$

Thus f is the desired $f_{\exists \mathrm{I}}$. ■

As its name suggests, the existence of the function $f_{\exists \mathrm{I}}$ justifies the existential introduction rule.

Lemma 2 For every $\exists x : A.\,P[x]$ there is a linear function $f_{\exists \mathrm{E}} : \exists x : A.\,P[x] \multimap A \star P[x]$ such that $f_{\exists \mathrm{E}}(\{x\} \times \alpha) = x \times \alpha$.

Proof. Similar to the above proof; pair $(x, a) \in \mathcal{A}_{\exists a:A.\,P[x]}$ with $(b, c) \in \mathcal{A}_{A\star P[x]}$ if and only if $b \in x$ and $a = c$. Then verify that this defines a linear function with the required property. ∎

The existence of $f_{\exists \mathrm{E}}$ justifies our existential elimination rule by showing that there is a natural linear function from existential pairs to strict pairs of certain types.

Lemma 3 For each $\alpha \in A$ there is a linear function $\pi_\alpha : \forall x : A.\, P[x] \multimap P[\alpha]$, such that

$$\begin{aligned} \pi_\alpha\left(\overset{\circ}{\bigcup_{x\in A'}} \{x\} \times \beta_x\right) &= \beta_\alpha \qquad \text{if } \alpha \in A' \\ &= \mathrm{Abort}_{P[\alpha]} \quad \text{otherwise.} \end{aligned}$$

Proof. Pair $(x, a) \in \mathcal{A}_{\forall x:A.\,P[x]}$ with $b \in \mathcal{A}_{P[\alpha]}$ if and only if $x = \alpha$ and $a = c$. The rest is checked as above. ∎

As its name suggests π_α is a projection function that extracts the component corresponding to α in any $s \in \forall x : A.\, P[x]$. The existence of such a projection for each $\alpha \in A$ justifies the universal elimination rule. Notice that even if $\alpha \times \beta_\alpha \in \forall x : A.\, P[x]$ and $\beta_\alpha \neq \mathrm{Abort}_{P[\alpha]}$, π_α could still return $\mathrm{Abort}_{P[\alpha]}$. This may seem strange, but in general the elements of the universal type can be thought of as partial functions. As with function types, the universal introduction rule derives a recipe for constructing an element of $P[x]$ from arbitrary x, so the *terms* of universal type in our term calculus correspond to maximal elements of the universal type in the coherence space semantics; we would not see this behavior from the universal elimination rule in practice.

2.2.3 Base Types

We finish off LCTT by providing two base types, the booleans and the natural numbers. The procedure for adding these types is straightforward and could be mimicked for other base types. We simply take a set of base type introduction and elimination rules that is reasonable for CTT, and "linearize" by imposing the typical null-intersection constraints on left sequents.

The rules for booleans are exactly the same as for CTT (see Table 1.3), except that in bool-E we require Γ and Δ be disjoint. Notice that although the constants True and False are not of banged type, they exist in the absence of any assumptions, and so they can be promoted and then used indefinitely.

The rules for the natural numbers are given in Table 2.7. They include

$$\frac{\vdash \text{True} : \text{bool}}{\vdash \text{Store}\,(\text{True}) : !\text{bool}} \qquad \frac{\vdash \text{False} : \text{bool}}{\vdash \text{Store}\,(\text{False}) : !\text{bool}}$$

Table 2.6: Promotion of boolean constants

$$\vdash 0{:}\mathbf{N} \quad (\mathrm{NI}_0) \qquad\qquad \frac{\Gamma \vdash n{:}\mathbf{N}}{\Gamma \vdash (\mathrm{succ}\, n){:}\mathbf{N}} \ (\mathrm{NI}_s)$$

$$\frac{\Gamma \vdash n{:}\mathbf{N} \qquad \Delta \vdash c{:}C[0/x] \qquad \Omega \vdash f : \forall n{:}\mathbf{N}.C[n/x] \multimap C[(\mathrm{succ}\, n)/x]}{\Gamma, \Delta, \Omega \vdash (\mathrm{prim}\, n\, c\, f) : C[n/x]} \ (\mathrm{NE})$$

Table 2.7: LCTT: rules for the naturals

NI$_0$: zero introduction.

NI$_s$: successor introduction. Given any natural number, under the same assumptions we can derive the successor of that natural number.

NE natural number elimination. This is an extremely important rule, because it allows us to do primitive recursion. The rule says that given some natural number n, the derivation of some property C for 0, and the derivation of a function which for any m will take a witness of $C[m/x]$ and produce a witness of $C[(\mathrm{succ}\, m)/x]$, we can produce a witness of $C[n/x]$. C will normally be some predicate parameterized by x, which states that x has some property. The special function prim is the *primitive recursor.* It iterates the function f n times on c to obtain the witness for $C[n/x]$. As was mentioned in Chapter 1, primitive recursion is equivalent to *a-priori* bounded tail recursion and hence to `for`-loops[27]. This is discussed in detail in Appendix 1.

Natural number elimination is unique among the base type elimination rules in that its arguments involve predicates over terms.

To reason about programs involving natural numbers, an essential addition is the $<$ predicate, whose introduction and elimination rules are shown in Table2.8.

The $<$I rule is actually a collection of rules, one for each $n \in \mathbf{N}$, that express the fact that any natural number is less than its successor. The elimination rules allow us to derive contradictions from derivations of $n < m$ together with $n = m$, and $n < m$ together with $m < n$, and express transitivity. We could obtain the same results by axiomatically

$$\vdash\ W_{<}(n) : (n < (\mathrm{succ}\ n)) \quad (<\mathrm{I})$$

$$\frac{\Gamma \vdash s : n < m \qquad \Delta \vdash t : n = m}{\Gamma, \Delta \vdash W_{<E_1}(s,t) : \bot}\ (<\mathrm{E}_=) \qquad \frac{\Gamma \vdash s : n < m \qquad \Delta \vdash t : m < n}{\Gamma, \Delta \vdash W_{<E_2}(s,t) : \bot}\ (<\mathrm{E}_<)$$

$$\frac{\Gamma \vdash s : n < m \qquad \Delta \vdash t : m < p}{\Gamma, \Delta \vdash W_{<T}(s,t) : n < p}\ (<\mathrm{E}_t)$$

Table 2.8: LCTT: rules for the < predicate

introducing the theorems

$$\begin{array}{l} \vdash\ \forall n{:}\mathbf{N}.\ \forall m{:}\mathbf{N}.\ (n < m) \sqcup (n = m) \sqcup (m < n) \\ \vdash\ \forall n{:}\mathbf{N}.\ \forall m{:}\mathbf{N}.\ (n < m) \star ((n = m) \sqcup (m < n)) \multimap \bot \quad \text{and} \\ \vdash\ \forall n{:}\mathbf{N}.\ \forall m{:}\mathbf{N}.\ \forall p{:}\mathbf{N}.\ (n < m) \star (m < p) \multimap (n < p) \end{array}$$

With these facts and the induction rule NE we can derive the usual host of useful identities, and define operations like addition and the predecessor (partial) function, but it would take considerable effort and the implementations of functions of computational interest (addition for instance) would be very inefficient. A more practical system needs to make all this available at the outset; this is discussed further in Chapter 4.

This completes the rules for LCTT, the linear logic equivalent of the constructive type theory CTT presented in Chapter 1. The intent remains the same: we use LCTT to develop fully-specified functions with type

$$\forall x{:}A.\ \exists y{:}B.\ P[x,y]$$

or, more generally,

$$\forall x{:}A.\ (G[x] \multimap \exists y{:}B.\ P[x,y])$$

where $G[x]$ is a *guard* predicate or precondition on x, asserting that it has some necessary property. The second form is the type of a fully-specified *partial* function from the terms of A to the terms of B. Unfortunately, LCTT is somewhat less convenient to use than CTT. Derivations tend to be longer and the fact that term names cannot be used repeatedly leads to a proliferation of assumption sequents $x{:}A \vdash x{:}A$ to set up uses of the ! elimination rules. Terms of banged type are reused by giving the different copies distinct names (dummy assumptions of the same type), each copy being used only once, and then collapsing the names together using contraction elimination. For example, a computation t that uses two

copies of $s\!:\!!\mathrm{N}$ must be developed as

$$\dfrac{s\!:\!!\mathrm{N} \vdash s\!:\!!\mathrm{N} \qquad \dfrac{\dfrac{x\!:\!!\mathrm{N} \vdash x\!:\!!\mathrm{N}}{\vdots} \qquad \dfrac{y\!:\!!\mathrm{N} \vdash y\!:\!!\mathrm{N}}{\vdots}}{x\!:\!!\mathrm{N}, y\!:\!!\mathrm{N} \vdash t\!:\!A}}{s\!:\!!\mathrm{N} \vdash \text{let } x = y = s \text{ in } t\!:\!A}$$

where the dummy names x and y are introduced to derive t and are then replaced with s using contraction. The original CTT is already a rather unfriendly environment, and it appears that LCTT makes a bad situation worse. In the next section, we discuss a simplification of LCTT that remedies this problem.

2.3 Modifications to Make LCTT More Convenient

The fact that identifiers can only be used once in LCTT terms leads to a rather tedious style of program development. Each time we need a copy of a term $s\!:\!!A$ we must introduce new dummy assumptions and use the !Ec rule to copy the term into them. At the intermediate stages in a derivation there may be several objects which are intended to be identical, and this makes derivations longer, harder to understand, and harder to write. If $s : !A$ is a reusable object, it seems that intuitively we should be able to use the identifier s repeatedly in a proof. In this section we propose a way to make a logic that is subsumed by LCTT but that has this convenience.

Suppose we have derived some LCTT sequent

$$\Gamma \vdash x\!:\!!A$$

of interest, and have also derived

$$x_1\!:\!!A, \ldots, x_n\!:\!!A, \Omega \vdash r\!:\!D$$

where Γ and $x_1\!:\!!A, \ldots, x_n\!:\!!A$, Ω have empty intersection. Suppose further we would like to eliminate all of the $x_1\!:\!!A, \ldots, x_n\!:\!!A$ with $x\!:\!!A$, deriving

$$x\!:\!!A, \Omega \vdash r'\!:\!D$$

where r' is the appropriate transformation of r arrived at by successive applications of !Ec.

The derivation is straightforward:

$$\dfrac{x_3'\!:!A \vdash x_3'\!:!A \qquad \dfrac{x_2'\!:!A \vdash x_2'\!:!A \qquad x_1\!:!A,\ \dots,\ x_n\!:!A,\Omega \vdash r\!:\!D}{x_2'\!:!A, x_3\!:!A,\ \dots,\ x_n\!:!A,\Omega \vdash \text{let } x_1 = x_2 = x_2' \text{ in } r\!:\!D}}{x_3'\!:!A, x_4\!:!A,\ \dots,\ x_n\!:!A,\Omega \vdash \begin{array}{l}\text{let } x_2' = x_3 = x_3' \text{ in}\\ \text{let } x_1 = x_2 = x_2' \text{ in } r\!:\!D\end{array}}$$

$$\vdots$$

$$\dfrac{\Gamma \vdash x\!:!A \qquad x_{n-1}'\!:!A, x_n\!:!A,\Omega \vdash \begin{array}{l}\text{let } x_{n-2}' = x_{n-1} = x_{n-1}' \text{ in}\\ \vdots\\ \text{let } x_1 = x_2 = x_2' \text{ in } r\!:\!D\end{array}}{\Gamma,\Omega \vdash \begin{array}{l}\text{let } x_{n-1}' = x_n = x \text{ in}\\ \vdots\\ \text{let } x_1 = x_2 = x_2' \text{ in } r\!:\!D\end{array}}$$

That is, we have introduced $n-2$ new assumptions and applied contraction elimination $n-1$ times, discharging the new assumptions in the process. We thus conclude a generalization of contraction elimination, namely

$$\dfrac{\Gamma \vdash x\!:!A \qquad x_1\!:!A,\ \dots,\ x_n\!:!A,\Omega \vdash r\!:\!D}{\Gamma,\Omega \vdash \text{let } x_1 =\ \dots\ = x_n = x \text{ in } r\!:\!D}\ \text{!EcG}$$

where Γ and $x_1\!:!A,\dots,x_n\!:!A,\Omega$ are disjoint and we have introduced an obvious equivalent syntax for the corresponding term.

Now suppose that we have developed derivations $D_1,\dots,D_n$ of sequents

$$\Gamma_1 \vdash s_1\!:\!B_1,\ \dots,\ \Gamma_n \vdash s_n\!:\!B_n$$

where the Γ_i are not all pairwise disjoint, but their pairwise intersections are all banged. Suppose further that we had intended to derive some sequent

$$!\Delta_1,\ \dots,\ !\Delta_k,\Omega \vdash r\!:\!D$$

but will not be able to complete the derivation because there will be linearity violations caused by the intersecting Γ_i. Then we claim that there is a systematic process by which we can repair our derivation.

Proof. An equivalent way of saying that the Γ_i are not all pairwise disjoint but have banged intersections is this is that there exists some collection of previously derived sequents

$$\{!\Delta_1 \vdash x_1\!:!A_1,\ \dots,\ !\Delta_k \vdash x_k\!:!A_k\}$$

where

1. the $!\Delta_l$ are pairwise disjoint

2. each sequent was used in more than one of the D_i, and

3. each Γ_i intersect Γ_j is a disjoint union of some subcollection of the $!\Delta_l$.

Actually, the right-hand sides need not be banged, but since the left-hand sides are we could use promotion at the end of the derivations of these sequents, and then insert uses of !E in the D_i where necessary, so we can assume this form without loss of generality. Knowing this, the process for repairing the derivation is

1. For each x_i, let $N(i)$ be the number of Γ_j of which $!\Delta_i$ is a subset–equivalently, the number of s_j in which x_i appears. Introduce $N(i)$ new assumption sequents

$$x_{i,1} : !A_i \vdash x_{i,1} : !A_i, \ \ldots, \ x_{i,N(i)} : !A_i \vdash x_{i,N(i)} : !A_i,$$

and use them to replace $!\Delta_i \ \vdash \ x_i : !A_i$ in each of the corresponding $N(i)$ affected derivations.

2. carry through the intended derivation of the $r : D$ sequent; the result will be

$$\begin{array}{l} x_{1,1} : !A_1, \ \ldots, \ x_{1,N(1)} : !A_1, \\ \ldots, \\ x_{k,1} : !A_k, \ \ldots, \ x_{k,N(k)} : !A_k, \Omega \vdash r' : D \end{array}$$

That is, each $!\Delta_l$ is replaced by $x_{l,1} : !A_l, \ \ldots, \ x_{l,N(l)} : !A_l$

3. apply the generalized contraction elimination rule (!EcG) k times, once for each of the x_l to eliminate the introduced

$$x_{l,1} : !A_l, \ \ldots, \ x_{l,N(l)} : !A_l,$$

finally obtaining

$$!\Delta_1, \ \ldots, \ !\Delta_k, \Omega \vdash \text{(nested let) in } r' : D$$

This gives us the desired result, though the intended term r (which would have had multiple references to each of the x_l and hence have been a nonlinear term) will be represented by an equivalent nested let construct.

This proves that if we modify LCTT so that $s : !A$ can be used repeatedly, for every derivation that we can do in the new logic there is an equivalent derivation in LCTT. In fact, we can allow any $s : A$, banged or not, to be reused, provided all the terms in the resulting left sequent after the derivation of $s : A$ are themselves banged. Any such term can be promoted to a banged term and then reused anyway, so the derivation could again be

repaired by using !E and !I where appropriate. We thus arrive at a new constructive type theory, which we will call SLCTT (semi-linear constructive type theory), obtained from LCTT using the following transformation:

1. allow named terms to persist for an entire proof; that is, once a term is introduced and given a name (by assumption or by derivation) it is remembered;

2. reinterpret the !E and !Ew rules not as destroying the original reference to the stored term, but as dereferencing it and doing nothing with it, respectively; and

3. allow the right side of any sequent with a banged left side to be reused within a derivation; we accomplish this by changing the inference rules to insist on *banged* intersections instead of empty intersections.

We can retain the !Ec rule so that we can make explicit copies of terms of banged type. In fact, with these developments and the fact that in LL $\vdash\ !A \multimap \mathbf{1} \sqcap A \sqcap (!A \star !A)$ we can simply replace the various ! elimination rules as shown in Table 2.9.

Rule	New Rule	Interpretation
!E	$\dfrac{\Gamma \vdash s:!A}{\Gamma \vdash \text{Get}\,(s):A}$	Dereference.
!Ec	$\dfrac{\Gamma \vdash s:!A}{\Gamma \vdash (s,s) : !A \star !A}$	Create two copies of s. Under the pointer-copy interpretation, the two components of the pair are aliases.
!Ew	$\dfrac{\Gamma \vdash s:!A}{\Gamma \vdash \text{Ignore}\,(s):\mathbf{1}}$	Ignore s for the moment; forget about in the current thread of computation because it can always be recovered by name later.

Table 2.9: SLCTT: New ! elimination rules.

SLCTT retains a certain amount of linearity for unbanged types, but allows banged (more generally, bang*able*) types to be treated as in regular CTT. As a programming language, SLCTT allows identifiers for bangable objects to appear multiple times in the same term, breaking the linearity of the term language, but the linearity can always be restored by introducing temporary variables and inserting Stores and Gets where necessary.

2.4 Summary

By imposing constraints on which objects can be repeatedly used, which ultimately appear as constraints on the intersections of sequent left-hand sides, linear logic gives us a way to do book-keeping on resources. In the type theory LCTT, the resources are expressions or terms in a programming language. Each term appears on the right-hand side of a sequent along with its type, and the left-context is the set of assumed terms that the term was constructed from. The assumed terms are fundamental and depend only on themselves. When a term t is used to produce a new term s, its left context is added to the left context of s, or replaces some part of it. Through their types, we can declare some fundamental terms as reusable, and some as single-use. The logic prevents us from using a single-use term more than once by checking for its illegal repeated occurrence in left contexts when they are combined.

As a programming system, LCTT offers no significant advantages over CTT, except possibly simplified storage allocation properties, which are not of direct interest to us. In fact, it has a number of disadvantages, inasmuch as the logic is more complicated and in a sense lower-level, requiring more inferences to prove equivalent things. However, in the next chapter we shall see that LCTT can be extended into a constructive type theory for an imperative programming language, complete with variables, arrays, and destructive updates. This extended system is the key contribution of this work.

Chapter 3

Mutable Storage

There are two fundamental prerequisites for correct reasoning about imperative programs. The first is *sequencing*. In any program there must be an understood order of evaluation of expressions which can be gleaned from the syntax of the language, at least for operations that update variables. If we do not know in what order updates will be performed, then we obviously cannot know with certainty what value a variable has at any given time[1]. The second is *interference control*. If two different expressions can modify the same storage, or *interfere*, we must know about it and take this into consideration in evaluating the effect of a sequence of operations.

In an imperative language, interfering expressions depend on or use the same variables, and this suggests that we might adapt LCTT to keep track of when objects interfere, by checking for their repeated occurrence in left-contexts. The obstacle to this is that variables are by nature reusable: they are named and can be read and written many times in a program, whereas LCTT polices only single-use objects. Moreover, LCTT has no notion of a reusable term that can change its meaning when it is used. A term $t :!A$ has a fixed meaning established when it is created; it is the symbolic constant of pure functional programming. LCTT also has no notion of explicit sequencing for uses of banged terms. If we make two copies of $t :!A$ and then use the copies in different computations, it makes no difference which computation is carried out first. It seems that variable types lie somewhere in between LCTT's banged and plain types. If we are to adapt LCTT to imperative programming, it appears that we need some kind of intermediate type.

[1]There are situations in which we allow the order of operations to be ambiguous. For example, consider a program with a graphical user interface that allows modification of a program option through one or more widgets. We cannot know the value of the option at any given time because we cannot predict the sequence of modifications of it. We can, however, reason about the option if we know it always has a value drawn from some predefined set, or that satisfies some invariant property. We will not consider this kind of system here, but will return to the notion of invariants for variables briefly in Chapter 6.

In this chapter we present a new binary connective $\triangleright$, called *before*, and a new storage modality $\dagger$, called the *regenerative* modality, and corresponding terms and inference rules for them. The $\triangleright$ connective was originally suggested by Girard, and the $\dagger$ modality by Reddy[20][21]. The new connective is somewhat like the multiplicative conjunction $\star$, but is interpreted as sequencing its left side before its right side. The new modality $\dagger$ is like ! in that it types reusable terms, but the uses of a $\dagger$ term must be sequenced using $\triangleright$. Associated with $\triangleright$ is some new structure in left contexts that allows a book-keeping of terms of $\dagger$ type.

Reddy developed $\dagger$ and combined it with $\triangleright$ in an effort to create a type system for imperative languages using linear logic as a foundation, and was able to type a fragment of Reynolds' interference-free idealized Algol with it. He found a way to type simple variables using $\dagger$, and exploited $\triangleright$ for sequencing and interference control.

Our innovations are to combine Reddy's system with LCTT, and extend the combination in several ways. We add term interpretations for $\triangleright$ and $\dagger$ that are compatible with the LCTT term language, and some inference rules that allow conversion from $\triangleright$ types to $\star$ types under special circumstances. The $\triangleright$ connective adds sequencing to LCTT, and the new rules in turn lead to "sequential" versions of many of the existing rules. We then identify sources of undesirable interference in the augmented LCTT, and elaborate an interference control strategy that exploits the new structural constraints on left-contexts. We develop extensions of Reddy's simple storage cells that model mutable records and arrays, and add specialized inference rules for the new types. We then develop this system into a full-blown constructive type theory by changing the rules for reading and updating variables and arrays so they return information on the effects of the operations. In the new system we are able to derive fully-specified imperative programs. Finally, we make a change to the book-keeping rules similar to the one made at the end of the last chapter, which allows identifiers for ! and $\dagger$ types to be freely reused. The final system is called ICTT, or Imperative Constructive Type Theory, and forms the basis for the implementation described in Chapters 4 and 5.

3.1 Foundations: Reddy's Linear Model of State

Reddy's system is an extension of intuitionistic linear logic which makes use of a new multiplicative connective $\triangleright$ ("before") that lies somewhere between $\star$ and $+$ in the coherence space semantics for LL, and a new modality $\dagger$, which he calls the *regenerative* modality. An intuitionistic sequent calculus for a fragment of LL extended with $\triangleright$ and $\dagger$ appears in Appendix C, along with a discussion of the semantic underpinnings of the new connectives.

Reddy's system is intuitionistic but not in the natural deduction style. We can obtain

natural-deduction style rules by treating the "R" rules as introduction rules, and then combining the "L" rules with Cut to obtain elimination rules. The result is shown in Table 3.1. We have also added a term assignment in each of the rules. The connective $\triangleright$ and modality $\dagger$ are related in much the same way as $\star$ and !, so the corresponding term assignments have a syntactic similarity. There are important semantic differences however: $\triangleright$ and $\dagger$ allow us to sequence computations with side effects in such a way that interference is controlled. Computationally, the expression $[s; t]$ is a *lazy* evaluation sequence: the value of t may depend on the value of s and is not fully determined unless s is discrete. Recall from Chapter 2 that a discrete type is one with a single, well-defined value like an natural number or boolean; a non-discrete type represents a collection of choices or a suspended computation like a lazy tuple or most functions. If s is non-discrete then it must be reduced to a discrete type before t can be used. As we will see, under these circumstances the lazy sequence $[s; t]$ can be converted into a strict sequence.

Left contexts are no longer multisets in Reddy's system, but partially ordered multisets. The syntax of a left context is now

$$\Gamma ::= \epsilon \mid A \mid \Gamma_0, \Gamma_1 \mid \Gamma_0; \Gamma_1$$

A sequent $\Gamma \vdash A$ can be interpreted as $\gamma \multimap A$ where γ is the formula obtained by replacing commas with $\star$ and semicolons with $\triangleright$. ϵ is the empty context and corresponds to $\mathbf{1}$. Each context Γ may be viewed as a pairing of

$|\Gamma|$, the multiset of formulas that appear in Γ; and

$\leq_\Gamma$, the partial order on the formulas of $|\Gamma|$, which is defined inductively as

$$\begin{aligned} \leq_A &= \{(A, A)\} \\ \leq_{\Gamma_0,\Gamma_1} &= \leq_{\Gamma_0} \cup \leq_{\Gamma_1} \\ \leq_{\Gamma_0;\Gamma_1} &= \leq_{\Gamma_0} \cup \leq_{\Gamma_1} \cup |\Gamma_0| \times |\Gamma_1| \end{aligned}$$

If A appears to the left of B in Γ with one or more intervening semicolons, then $A \leq_\Gamma B$. Formulas which are separated only by commas are unrelated in $\leq_\Gamma$. As usual, differently named sequents Γ and Δ do not intersect[2]. In the expression $\Delta[\bullet]$, $[\bullet]$ represents a *hole*, a subcontext of Δ of a particular form. The expression $\Delta[\Gamma]$ is the sequent obtained from Δ by deleting the contents of a hole and inserting Γ in its place.

The new inference rules include

[2]Syntactically identical formulas are considered different instances of the same formula; when terms accompany formulas, the instances of a formula are distinguished by different terms.

$\star$Eser$_1$: $\star$ serial elimination number one. This rule states that a multiplicative conjunction can eliminate two compatible component types and an intervening semicolon. This is because there is an injection from $A \star B$ to $A \triangleright B$—a strict pair can be thought of as a lazy sequence in which the sequencing is unnecessary;

$\star$Eser$_2$: $\star$ serial elimination number two. A multiplicative conjunction can eliminate compatible component types in the reverse order provided they are separated by a comma. There is no corresponding rule if they are separated by a semicolon. This rule asserts that in any left context permuting the independent subcontexts (collections of consecutive typed-terms separated only by commas) does not change the meaning of the context;

$\triangleright$I: $\triangleright$ and $\dagger$ have an analogous relationship to that between $\star$ and !; this rule is like $\star$I, but for $\triangleright$;

$\triangleright$E: the same as $\star$E, but for $\triangleright$;

$\dagger$I: the same as !I, but for $\dagger$. Some objects in Γ can be of ! type because, just as $A \star B$ can be treated as an $A \triangleright B$, a $!A$ can be treated as a $\dagger A$;

$\dagger$E: analogous to !E;

$\dagger$Ew: analogous to !Ew;

$\dagger$Et: analogous to !Ec, but called *threading* instead of contraction, because of the understood left-to-right order;

!Eser: as was said in the description of $\dagger$I, a $!A$ can be treated as a $\dagger A$. The special function Convert may be viewed as a type cast from !-storage to $\dagger$-storage. Although we use $\dagger$ to build types for variables and arrays, we disallow the equivalent ! types, so there is no danger of plugging an immutable object in place of a mutable one and then attempting to change its value.

We may view a context with commas and semicolons as a semicolon-separated sequence of independent subcontexts. Since $\star$ is commutative, within each independent subcontext the formulas may be permuted without changing the meaning of the whole, but since $\triangleright$ is not commutative, we may not exchange formulas on either side of a semicolon. The serialization rule from Appendix C

$$\frac{\Gamma \vdash A}{\Gamma' \vdash A} \;(\text{Ser}) \qquad |\Gamma'| = |\Gamma| \text{ and } \Gamma'_{\leq} \subseteq \Gamma_{\leq}$$

$$\frac{\Gamma \vdash A \star B \qquad \Delta[x:A;y:B] \vdash t:C}{\Delta[\Gamma] \vdash \text{let } (x,y) = s \text{ in } t : C}\ (\star\text{Eser}_1)$$

$$\frac{\Gamma \vdash A \star B \qquad \Delta[x:B,y:A] \vdash t:C}{\Delta[\Gamma] \vdash \text{let } (y,x) = s \text{ in } t : C}\ (\star\text{Eser}_2)$$

$$\frac{\Gamma \vdash s:A \qquad \Delta \vdash t:B}{\Gamma;\Delta \vdash [s;t] : A \rhd B}\ (\rhd\text{I})$$

$$\frac{\Gamma \vdash s:A \rhd B \qquad \Delta[x:A;y:B] \vdash t:C}{\Delta[\Gamma] \vdash \text{let } [x;y] = s \text{ in } t : C}\ (\rhd\text{E})$$

$$\frac{\Gamma \vdash s:A}{\Gamma \vdash \text{RStore}\,(s) \ : \ \dagger A}\ (\dagger\text{I})$$

if Γ is an independent context with only ! and † formulas

$$\frac{\Gamma \vdash s:\dagger A \qquad \Delta[x:A] \vdash t:B}{\Delta[\Gamma] \vdash \text{let } x = \text{RGet}\,(s) \text{ in } t : B}\ (\dagger\text{E}) \qquad \frac{\Gamma \vdash s:\dagger A \qquad \Delta[\epsilon] \vdash t:B}{\Delta[\Gamma] \vdash \text{RIgnore}\,(s)\,;\, t : B}\ (\dagger\text{Ew})$$

$$\frac{\Gamma \vdash s:\dagger A \qquad \Delta[x:\dagger A;y:\dagger A] \vdash t:B}{\Delta[\Gamma] \vdash \text{let } x = y = s \text{ in } t : B}\ (\dagger\text{Et})$$

$$\frac{\Gamma \vdash s:!A \qquad \Delta[x:\dagger A] \vdash t:B}{\Delta[\Gamma] \vdash \text{let } x = \text{Convert}\,(s) \text{ in } t : B}\ (!\text{Eser})$$

Table 3.1: Natural-deduction version of Reddy's rules

can be viewed as the pair of rules

$$\frac{\Gamma[A; B] \vdash C}{\Gamma[A, B] \vdash C}\ (\mathrm{Ser}_1) \qquad \frac{\Gamma[B, A] \vdash C}{\Gamma[A, B] \vdash C}\ (\mathrm{Ser}_2)$$

which allow semicolons to be converted to commas and formulas in independent subcontexts to be permuted. The elimination rules $\star\mathrm{Eser}_1$ and $\star\mathrm{Eser}_2$ are derived from Ser_1 and Ser_2 using $\star$E (see Appendix C). They reflect the existence of a linear injection from $A \star B$ to $A \triangleright B$ and a linear isomorphism between $A \star B$ and $B \star A$.

We add $\dagger$, $\triangleright$ and the above rules to the original LCTT rules and will use them to support mutable storage objects. The order in which argument contexts are glued together is now important, however, and some of the original LCTT elimination rules must be converted to hole-filling forms. The required new versions are shown in Table 3.2. The expression $[x : A], \Delta$ is called a context with a *left hole*, and indicates that $x : A$ is at the far left of its context, separated from the rest (Δ) by a comma, or is in some left independent subcontext with at least two elements and can be moved to the far left via permutation. We can define a context with a *right hole* analogously. The $\multimap$E rule also appears in the table, but with the argument sequent placed before the function sequent. This is consistent with a strict evaluation semantics in which the argument (with any side-effects it may have) is evaluated first and the result is fed into the function.

3.1.1 Storage Cells and Rules for Eliminating $\triangleright$

Reddy proposed that a storage cell for a value of (discrete) type A could be modeled by the regenerative type

$$\dagger\,(A \sqcap {\sim}A)$$

where ${\sim}A := A \multimap \mathbf{1}$. The first component of the additive disjunction represents a *get* or read operation, and the second component represents a *put* or write operation. There is a minor shortcoming the rules in Table 3.1, we would like the terms for finished derivations to be strict, so all lazy sequences must be reduced to strict ones, but $\triangleright$E by itself is insufficient for this. Thus we need some additional rules to help remove $\triangleright$ on the right side of a turnstile. For example, suppose that we have a natural number base type $\mathbf{N}$, and that we wish to derive a program fragment that increments the value in a $\mathbf{N}$ storage cell. The only way to identify two terms of type $\dagger\,(\mathbf{N} \sqcap {\sim}\mathbf{N})$ is by using the $\dagger$Et rule, which requires that the terms be related in the partial order. Let $\mathbf{N}_c := \dagger\,(\mathbf{N} \sqcap {\sim}\mathbf{N})$, and define the terms

$$\begin{aligned} {*v} &:= \pi_1\,(\mathrm{RGet}\,(v)) : A \\ (v := \_) &:= \pi_2\,(\mathrm{RGet}\,(v)) : {\sim}A \end{aligned}$$

$$\frac{\Gamma \vdash s : A \star B \qquad \Delta[x{:}A, y{:}B] \vdash t{:}C}{\Delta[\Gamma] \vdash \text{let } (x, y) = s \text{ in } t : C} \ (\star\mathrm{E})$$

$$\frac{\Delta \vdash s : A \sqcup B \qquad \begin{array}{c}[x{:}A], \Gamma \vdash t{:}C \\ [y{:}B], \Gamma \vdash t'{:}C\end{array}}{\Delta, \Gamma \vdash \text{case } s\ t\ t' : C} \ (\sqcup\mathrm{E})$$

$$\frac{[x{:}A], \Gamma \vdash t{:}B}{\Gamma \vdash \lambda x.t : A \multimap B} \ (\multimap \mathrm{I})$$

$$\frac{\Delta \vdash t{:}A \qquad \Gamma \vdash s : A \multimap B}{\Delta, \Gamma \vdash s\,t : B} \ (\multimap \mathrm{E})$$

$$\frac{\Gamma \vdash s{:}!B \qquad \Delta[x{:}B] \vdash t{:}A}{\Delta[\Gamma] \vdash \text{let } x = \text{Get}\,(s) \text{ in } t : A} \ (!\mathrm{E})$$

$$\frac{\Gamma \vdash s{:}\,!B \qquad \Delta[x{:}\,!B, y{:}\,!B] \vdash t{:}A}{\Delta[\Gamma] \vdash \text{let } x = y = s \text{ in } t : A} \ (!\mathrm{Ec})$$

Table 3.2: New hole-filling rules adapted from LCTT.

where we have written the assignment function in infix notation, with an underscore indicating the place for the argument. These are the terms that can be obtained from $v : \dagger(A \sqcap {\sim}A)$ by using $\dagger$E and then $\sqcap\mathrm{E}_1$ and $\sqcap\mathrm{E}_2$, respectively. Then we could proceed as

$$\dfrac{\dfrac{\dfrac{\dfrac{c : \mathbf{N}_c \vdash c : \mathbf{N}_c}{\vdots}}{c : \mathbf{N}_c \vdash *c : \mathbf{N}} \qquad \vdash 1 : \mathbf{N}}{c : \mathbf{N}_c \vdash *c + 1 : N} \qquad \dfrac{\dfrac{d : \mathbf{N}_c \vdash d : \mathbf{N}_c}{\vdots} \qquad \dfrac{\epsilon : \mathbf{N}_c \vdash \epsilon : \mathbf{N}_c}{\vdots}}{d : \mathbf{N}_c;\ \epsilon : \mathbf{N}_c \vdash [d := \_;\ \epsilon] : {\sim}\mathbf{N} \triangleright \mathbf{N}_c}}{\begin{array}{l} c : \mathbf{N}_c;\ d : \mathbf{N}_c;\ \epsilon : \mathbf{N}_c \ \vdash \ [*c + 1; \\ \qquad\qquad\qquad\qquad [d := \_;\ \epsilon]] : \mathbf{N} \triangleright ({\sim}\mathbf{N} \triangleright \mathbf{N}_c)\end{array}}$$

assuming that our λ-calculus has been extended with a suitable addition function $+$ of type $\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$, here written using infix notation to improve readability[3]. But this is as far as we can get, setting up the sequence of natural number, function to assign it to a cell, and new storage cell. We cannot finish the computation because there is no rule to perform the necessary function application, or to discard its result.

We need additional rules that eliminate $\triangleright$ on the right of a sequent under certain cir-

[3]Note that the proof tree takes a shortcut with the application of the addition function: in a full tree, we would introduce the function through some sort of special rule and then apply $\multimap$E twice, but this is longer and harder to read. The intent should be clear.

cumstances. We obtain the first such rule by observing that when the type A is discrete, $\sim_{A \triangleright B}$ becomes identical to $\sim_{A \star B}$, and so there is a linear injection from $A \triangleright B$ to $A \star B$, and a corresponding rule

$$\frac{\Gamma \vdash s : A \triangleright B}{\Gamma \vdash \text{let } [x; y] = s \text{ in letemp } t_1 = x \text{ in } (t_1, y) : A \star B} \ (\triangleright\text{To}\star) \quad \text{provided } A \text{ is discrete.}$$

where **letemp** is a special form of **let** that is guaranteed not to be removed or duplicated when the term is compiled to an executable language. In the resulting term the first component of the lazy sequence is extracted and its evaluation forced before the second component, thus preserving the left-to-right evaluation order implicit in the sequence. This is critical, because $A \star B$ and $B \star A$ are isomorphic; from a strict product we can derive a term which exchanges components in the product. If the final term were simply let $[x; y] = s$ in (x, y) with an understood order of evaluation, then applying such a transformation could change the order of evaluation, and the sequential semantics of $[x; y]$ would be lost. The given form ensures that the reduced forms of the components may be used in any order, but the reductions stay correctly sequenced.

The $\triangleright$To$\star$ rule is fundamental, because we can use it to justify "sequential" versions of many of the original LCTT rules.

Proposition 1 *Every rule of the form*

$$\frac{\Gamma_1 \vdash s_1 : A_1 \quad \ldots \quad \Gamma_n \vdash s_n : A_n}{\Gamma_1, \ldots, \Gamma_n \vdash \mathrm{E}(s_1, \ldots, s_n) : \mathrm{T}(A_1, \ldots, A_n)} \ (\mathrm{R})$$

where $E(s_1, \ldots, s_n)$ is a term construction on $s_1, \ldots, s_n$ and $T(A_1, \ldots, A_n)$ is a type construction on $A_1, \ldots, A_n$, together with $\star E$ derives a rule

$$\frac{\Gamma \vdash s : A_1 \star \ldots \star A_n}{\Gamma \vdash \text{let } (s_1, \ldots, s_n) = s \text{ in } E(s_1, \ldots, s_n) : T(A_1, \ldots, A_n)} \ (\star\mathrm{R})$$

Proof. The derivation requires only the original rule R and an application of (generalized) $\star$E:

$$\frac{\Gamma \vdash s : A_1 \star \ldots \star A_n \quad \dfrac{s_1 : A_1 \vdash s_1 : A_1 \quad \ldots \quad s_n : A_n \vdash s_n : A_n}{s_1 : A_1, \ldots, s_n : A_n \vdash \mathrm{E}(s_1, \ldots, s_n) : \mathrm{T}(A_1, \ldots, A_n)}}{\Gamma \vdash \text{let } (s_1, \ldots, s_n) = s \text{ in } \mathrm{E}(s_1, \ldots, s_n) : \mathrm{T}(A_1, \ldots, A_n)}$$

∎

Corollary 1 *For discrete $A_1, \ldots, A_{n-1}$ there is a sequential version of R given by*

$$\frac{\Gamma_1 \vdash s_1 : A_1 \quad \ldots \quad \Gamma_n \vdash s_n : A_n}{\Gamma_1; \ldots; \Gamma_n \vdash \begin{array}{l} \text{letemp } t_1 = s_1 \text{ in} \\ \vdots \\ \text{letemp } t_{n-1} = s_{n-1} \text{ in} \\ \mathrm{E}(t_1, \ldots, t_{n-1}, s_n) : \mathrm{T}(A_1, \ldots, A_n) \end{array}} \quad (\mathrm{R}_S)$$

and R_S *together with (generalized)* $\triangleright$E *derives a sequential counterpart*

$$\frac{\Gamma \vdash s : A_1 \triangleright \ldots \triangleright A_n}{\Gamma \vdash \begin{array}{l} \text{let } [s_1; \ldots; s_n] = s \text{ in} \\ \text{letemp } t_1 = s_1 \text{ in} \\ \vdots \\ \text{letemp } t_{n-1} = s_{n-1} \text{ in} \\ \mathrm{E}(t_1, \ldots, t_{n-1}, s_n) : \mathrm{T}(A_1, \ldots, A_n) \end{array}} \quad (\triangleright\mathrm{R}_S)$$

to $\star$R.

Proof. Construct $\Gamma_1; \ldots; \Gamma_n \vdash [s_1; \ldots; s_n] : A_1 \triangleright \ldots \triangleright A_n$ from the $\Gamma_i \vdash s_i : A_i$ using (generalized) $\triangleright$I and then use (generalized) $\triangleright$To$\star$ to obtain

$$\Gamma_1; \ldots; \Gamma_n \vdash \begin{array}{l} \text{letemp } t_1 = s_1 \text{ in} \\ \vdots \\ \text{letemp } t_{n-1} = s_{s-1} \text{ in} \\ (t_1, \ldots, t_{n-1}, s_n) : A_1 \star \ldots \star A_n \end{array}$$

The first result is then obtained after an application of $\star$R and simplifying the **let** expression it introduces. The proof of the second result is analogous to the proof of Lemma 1 except R is replaced by SR and $\star$E is replaced by $\triangleright$E. ∎

A frequently-used sequentialized rule is $\multimap \mathrm{E}_S$, the rule for sequential function application. This is

$$\frac{\Gamma \vdash s : A \qquad \Delta \vdash f : A \multimap B}{\Gamma; \Delta \vdash \begin{array}{l} \text{letemp } t_1 = s \text{ in} \\ (f\, t_1) : B \end{array}} \quad (\multimap \mathrm{E}_S) \quad \text{provided } A \text{ is discrete}$$

The corresponding $\triangleright \multimap \mathrm{E}_S$ will be of use in completing our derivation; we will give it the less cryptic name SApply, for "sequential application."[4]

If A is a single-atom type and has only the witnesses Abort_A and W_A, then we will sometimes write

$$\text{let } [x; y] = s \text{ in } x; (W_A, y)$$

[4]The corresponding $\star \multimap$E rule is Apply and takes an ordered pair with an argument and a function and applies the function to the argument, but without concern for sequencing.

instead of

$$\text{let } [x; y] = s \text{ in letemp } t_1 = x \text{ in } (t_1, y)$$

as the resulting term for $\triangleright$To$\star$. The version of $\triangleright$To$\star$ with this alternative term is called $\triangleright$To$\star_{SA}$. We have here generalized the strict sequence notation to include a left side of any single-atom type, rather than just $\mathbf{1}$. Since A is single-atom, it has only the witnesses Abort$_A$ and W_A; if x is or reduces to Abort$_A$ the computation will abort anyway, so there is no problem pairing W_A with r as the result. The $\triangleright$To$\star_{SA}$ version allows a more familiar syntax in the event that the term for the A component is an operation with a side effect, such as assignment. For Reddy's storage cells $x := t$ has type $\mathbf{1}$ and the new strict sequence is not an extension of the old, but in the sections covering specification we will change the type of assignment statements and the extension will be necessary.

The second rule we require is obtained from Reddy's observation that if A is a discrete type the identity function on $\mathcal{A}_A \times \mathcal{A}_{\mathbf{1}} \times \mathcal{A}_B$ and the projection isomorphism $\pi_2 : \mathcal{A}_{\mathbf{1}} \times \mathcal{A}_B \to \mathcal{A}_B$ induce a linear injection from $(A \multimap \mathbf{1}) \triangleright B$ to $A \multimap B$, so that $(A \multimap \mathbf{1}) \triangleright B$ is a subset of the function space $A \multimap B$. Thus every term of type $(A \multimap \mathbf{1}) \triangleright B$ has a corresponding term of type $A \multimap B$. This justifies a rule

$$\frac{\Gamma \vdash s : (A \multimap \mathbf{1}) \triangleright B}{\Gamma \vdash \lambda x.\, \text{let } [f; r] = s \text{ in } (f\, x);\ r : A \multimap B} \ (\text{Sfunc}) \quad \text{provided } A \text{ is discrete.}$$

The lazy sequence s is decomposed to produce a function term $f : A \multimap \mathbf{1}$ and the result part r. Then f is applied to the argument x, producing a result of type $\mathbf{1}$. From the left-to-right communication semantics of $\triangleright$, the meaning of r may depend on the computation $f\,x$; for purposes this communication will take the form of side effects produced by evaluating $f\,x$. As with $\triangleright$To$\star$, this argument can be generalized to $(A \multimap S) \triangleright B$ for any discrete type A and single-atom type S. The resulting rule is

$$\frac{\Gamma \vdash s : (A \multimap S) \triangleright B}{\Gamma \vdash \lambda x.\, \text{let } [f; r] = s \text{ in } (f\, x);\ (W_S, r) : A \multimap (S \star B)} \ (\text{Sfunc}_{SA})$$

provided A is discrete and S is single-atom

With these rules our derivation can be completed using $\triangleright$I, Sfunc, $\triangleright$I again, and Sapply as

follows:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{c : \mathrm{N}_c \;\vdash\; c : \mathrm{N}_c}{\vdots}
}{c : \mathrm{N}_c \;\vdash\; *c : \mathrm{N}} \qquad \vdash\; 1 : \mathrm{N}
}{
\begin{array}{c} c : \mathrm{N}_c \;\vdash\; *c + 1 : N \\ \\ \vdots \end{array}
}
\qquad
\dfrac{
\dfrac{
\dfrac{d : \mathrm{N}_c \;\vdash\; d : \mathrm{N}_c}{\vdots} \qquad \dfrac{\epsilon : \mathrm{N}_c \;\vdash\; \epsilon : \mathrm{N}_c}{\vdots}
}{d : \mathrm{N}_c;\; \epsilon : \mathrm{N}_c \;\vdash\; [d := \_;\; \epsilon] : {\sim}\mathrm{N} \rhd \mathrm{N}_c}
}{
\begin{array}{l} d : \mathrm{N}_c;\; \epsilon : \mathrm{N}_c \;\vdash \\ \qquad \lambda x.\,\mathrm{let}\ [f;\ r] = [d := \_;\ \epsilon]\ \mathrm{in} \\ \qquad\qquad (f\,x);\ r : \mathrm{N} \multimap \mathrm{N}_c \end{array}
}
}{
\dfrac{
\begin{array}{ll} c : \mathrm{N}_c;\; d : \mathrm{N}_c;\; e : \mathrm{N}_c \;\vdash & [*c + 1; \\ & \lambda x.\,\mathrm{let}\ [f;\ r] = [d := \_;\ e]\ \mathrm{in} \\ & \quad (f\,x);\ r] : \mathrm{N} \rhd (\mathrm{N} \multimap \mathrm{N}_c) \end{array}
}{
\begin{array}{ll} c : \mathrm{N}_c;\; d : \mathrm{N}_c;\; e : \mathrm{N}_c \;\vdash & \mathrm{letemp}\ t_1 = *c + 1\ \mathrm{in} \\ & \mathrm{let}\ [f;\ r] = [d := \_;\ e]\ \mathrm{in} \\ & (f\,t_1);\ r : \mathrm{N}_c \end{array}
}
}
$$

and given a storage cell of interest $\Gamma \vdash s : \dagger(\mathrm{N} \sqcap {\sim}\mathrm{N})$ we can use †Et to obtain

$$
\begin{array}{ll} s :\ \dagger(\mathrm{N} \sqcap {\sim}\mathrm{N}) \;\vdash & \mathrm{let}\ c = d = e = s\ \mathrm{in} \\ & \mathrm{letemp}\ t_1 = *c + 1\ \mathrm{in} \\ & \mathrm{let}\ [f;\ r] = [d := \_;\ \epsilon]\ \mathrm{in} \\ & (f\,t_1)\,;\ r :\ \dagger(\mathrm{N} \sqcap {\sim}\mathrm{N}) \end{array}
$$

Carrying through the substitutions for the lets we see that the term is equivalent to

$$(s := *s + 1)\ ;\ s.$$

In a typical imperative programming language like C this would be

```
s = s + 1;
s
```

The new $\rhd$ elimination rules are summarized in Table 3.3. We call LCTT extended with Reddy's rules and the new rules for eliminating $\rhd$ LRCTT.

3.1.2 Interference Control

LRCTT has an imperative term language, and so we must now be concerned with the possibility that two terms modify the same storage location. Terms that reference the same storage location are said to *interfere*. Interference is a necessary part of imperative programming, but it can make reasoning about what a program does more difficult. This problem is especially bad if there is unrestricted *aliasing*, because an update of one variable could mean that what was known about the contents of some other variable is no longer true.

$$\frac{\Gamma \vdash s : A \triangleright B}{\Gamma \vdash \text{let } [x; y] = s \text{ in letemp } t_1 = x \text{ in } (t_1, y) : A \star B} \quad (\triangleright\text{To}\star)$$

provided A is discrete

$$\frac{\Gamma \vdash s : A \triangleright B}{\Gamma \vdash \text{let } [x; y] = s \text{ in } x; (W_A, y) : A \star B} \quad (\triangleright\text{To}\star_{SA})$$

provided A is single-atom

$$\frac{\Gamma \vdash s : (A \multimap \mathbf{1}) \triangleright B}{\Gamma \vdash \lambda x.\, \text{let } [f; r] = s \text{ in } (f\, x); r : A \multimap B} \quad (\text{Sfunc})$$

provided A is discrete

$$\frac{\Gamma \vdash s : (A \multimap S) \triangleright B}{\Gamma \vdash \lambda x.\, \text{let } [f; r] = s \text{ in } (f\, x);\ (W_S, r) : A \multimap (S \star B)} \quad (\text{Sfunc}_{SA})$$

provided A is discrete and S is single-atom

$$\frac{\Gamma_1 \vdash s_1 : A_1 \quad \ldots \quad \Gamma_n \vdash s_n : A_n}{\begin{array}{ll}\Gamma_1; \ldots; \Gamma_n \vdash & \text{letemp } t_1 = s_1 \text{ in} \\ & \vdots \\ & \text{letemp } t_{n-1} = s_{n-1} \text{ in} \\ & \mathrm{E}(t_1, \ldots, t_{n-1}, s_n) : \mathrm{T}(A_1, \ldots, A_n)\end{array}} \quad (\text{SR})$$

provided $A_1, \ldots, A_{n-1}$ are discrete

$$\frac{\Gamma \vdash s : A_1 \triangleright \ldots \triangleright A_n}{\begin{array}{ll}\Gamma \vdash & \text{let } [s_1; \ldots; s_n] = s \text{ in} \\ & \text{letemp } t_1 = s_1 \text{ in} \\ & \vdots \\ & \text{letemp } t_{n-1} = s_{n-1} \text{ in} \\ & \mathrm{E}(t_1, \ldots, t_{n-1}, s_n) : \mathrm{T}(A_1, \ldots, A_n)\end{array}} \quad (\triangleright\text{SR})$$

provided $A_1, \ldots, A_{n-1}$ are discrete

Table 3.3: additional $\triangleright$ elimination rules

Unfortunately, the linearity of LRCTT forces us to use a separate name for each use of a reusable object (those objects of ! or † type), and does not automatically prevent us from developing programs in which terms interfere in undesirable ways. In this section we develop a strategy for controlling aliasing and preventing undesirable interference.

Definitions of Interference and Aliasing

We begin with formal definitions of interference and aliasing in the derivation of a sequent $\Gamma \vdash a : A$. We assume that

1. in advance of the derivation, we decide upon a collection of distinct mutable storage objects called the *state* of the derivation;

2. with each term $x : A$ we associate a subset of the state, ms (x), the collection of mutable storage objects referenced by x;

3. by extension, with each context Γ we associate a set ms (Γ) defined by

$$\bigcup_{x \in |\Gamma|} \text{ms } (x) \, ;$$

4. the assignment of the **ms** sets must be consistent within sequents; that is, if the sequent $\Delta \vdash x : A$ appears in the derivation, then ms $(\Delta) =$ ms (x); and

5. the assignment of the **ms** sets must be consistent with the way hole-filling rules are used in the derivation; *i.e.* if Γ is used to replace a hole Δ_h in Δ, then we must have ms $(\Delta_h) =$ ms (Γ).

Definition 1 We say $a : A$ *interferes with* $b : B$ iff ms $(a) \cap$ ms $(b) \neq \emptyset$. A term $a : A$ for which ms $(a) = \emptyset$ is called *applicative*; a term which is not applicative is called *nonapplicative*. Similarly, we say context Γ interferes with context Δ iff ms $(\Gamma) \cap$ms $(\Delta) \neq \emptyset$.

On the collection of nonapplicative terms in the derivation the interferes-with relation is obviously reflexive and symmetric. With each mutable storage object γ in the state we associate a unique type A_γ; we restrict all such types to be † types. For each γ the collection of terms

$$\{a \in A_\gamma : a \text{ appears in the derivation and ms } (a) = \{\gamma\}\}$$

is called the *aliases* of γ. We say distinct $a, b \in$ aliases (γ) are aliases, or alias one another. It should be obvious that the restriction of the interferes-with relation to any union of alias sets is an equivalence relation[5] whose equivalence classes are the alias sets themselves.

So far we have defined aliasing only for mutable storage locations—*i.e.* various kinds of storage cells. The notion of aliasing can be extended to terms which do not directly correspond to storage locations, and even to terms which are applicative. For example, a reusable procedure[6] $f : \dagger(A \multimap B)$ may have aliases. For each type appearing in the derivation we decide in advance on an equivalence relation, called the alias relation, over the terms of that type; the equivalence classes of the relation are called *alias sets*. The alias relations must be defined so that

- if $a : A$ aliases $b : A$ then ms $(a) =$ ms (b), but not necessarily the converse;
- all applications of !Ec or †Et to terms of type A merge aliases;
- for each alias set S, there is at most one $x : A \in S$ which is not introduced by assumption; that is, at most one $x : A$ which is derived from other terms and thus appears only on the right in some sequent $\Delta \vdash x : A$; such an $x : A$ is the preferred representative of the alias set.

When we need to draw a distinction we will refer to aliasing of storage objects as *storage aliasing*, and to the alias sets of storage objects as *storage alias sets*. The (disjoint) union of alias relations over all the types appearing in the derivation is simply called *the* alias relation for the derivation.

Undesirable Interference

Those LRCTT inference rules that produce new contexts from old ones can be grouped into three categories: those rules that paste together old contexts using commas; those rules that paste together old contexts using semicolons; and those rules that fill a hole in one context with another context. If the argument contexts interfere, the comma rules are frequently unsuitable because they lead to terms whose side effects are ambiguous. For

[5]That is, a reflexive, symmetric and transitive relation, like equality. An equivalence relation on a set S partitions S into disjoint subsets, called *equivalence classes*. Each pair of elements in the same equivalence class is related; no two elements from distinct equivalence classes are related.

[6]In LCTT, a function can only be used once unless its type is banged $!(A \multimap B)$. In LCTT with Reddy's extensions, there are two reusable function types, $!(A \multimap B)$ and $\dagger(A \multimap B)$. The second type must be used if the function depends on one or more terms of † type. A function that reads or modifies a global mutable object is an example of such a function. We will sometimes call such functions *procedures*.

example, consider the $\star$I rule

$$\frac{\Gamma \vdash s : A \qquad \Delta \vdash t : B}{\Gamma, \Delta \vdash (s,t) : A \star B}$$

If s and t interfere (iff Γ and Δ interfere) and both update the storage they depend on, then it is not clear what the effect of (s,t) is because there is no preferred order of evaluation of s and t. We are not free to impose one based on, for example, left-to-right order in the tuple, because (s,t) and (t,s) are isomorphic. Another example is the $\multimap$ E rule:

$$\frac{\Gamma \vdash s : A \qquad \Delta \vdash f : A \multimap B}{\Gamma, \Delta \vdash (f\,s) : B}$$

Here f should not in general interfere with s. This is not to say that a function cannot modify its argument if it is mutable, but rather that if it modifies its argument, a global variable that the function depends on is not modified as a consequence, and vice-versa. If we permit this kind of interference, the effect of a function application is no longer a simple composition of the effect of its argument (if any) and the effect of its body. For example, consider:

```
let y : N var in
⋮
 λx.  y := (*y) + 1;
      x := (*x) + (*y);
      x :
N var ⊸ N var
```

For almost all arguments x the function has the effect of adding $y+1$ to x and leaving y alone. But if x is y, then the effect on x is to add $y+2$ to it; and y has been incremented and doubled. One can imagine a system in which uncontrolled interference is allowed in this case[7], but its possibility must be acknowledged in any accompanying reasoning by considering the case that x and y are aliases for the same storage. Moreover, curried functions that permit interference between the argument and the function body are equivalent to multi-argument functions that permit interference between arguments. For functions with several mutable arguments of the same type (or a variable and an array argument over the same type), we must then consider all possible combinations of interference between arguments, which clearly leads to a combinatorial explosion and a potentially huge increase in the size of any statement of effect. Since

$$\lambda x.\,\lambda y.t : A \multimap B \multimap C$$

[7]This is not possible in general; if a language has some equivalent of updatable pointers or references then what is aliased when the program runs can be a function of the input. Clearly, a program cannot anticipate all possible inputs to itself unless they form a finite set.

is supposed to be isomorphic to

$$\lambda z : \text{let } (x, y) = z \text{ in } t : (A \star B) \multimap C$$

according to the typing rules, insisting that we permit interference between a function body and its argument is insisting that we permit interference between components of a strict tuple.

Because of the $\triangleright$To$\star$ rule and the SR metarule, for each rule that introduces a comma there is a corresponding rule that introduces a semicolon instead. In contrast to the comma rules, these rules are better suited to interfering arguments because they enforce an order of evaluation. They also restrict all terms except the last to be discrete, and this *usually* solves the problem encountered with $\multimap$ E above, in the following sense. Recall that all updatable storage types are $\dagger$ types, and these are never discrete. Most discrete types encountered in practice are constructions over base types using $\star$ and $\sqcup$, so once evaluated[8] they represent passive data and there can be no unexpected side-effects on or through the use of such objects in a function body. However, there are exceptional objects of discrete type that do have side effects. An example is the function

$$\lambda x.x; s := 10 : \mathbf{1} \multimap \mathbf{1}$$

where s is a global storage cell of type $\dagger(\mathbf{N} \sqcap (\mathbf{N} \multimap \mathbf{1}))$. Since $\mathbf{1}$ is single-atom, the domain type is single-atom and the range type is discrete, so this function is of discrete type. Obviously, s will appear in the context of this term; using it as an argument to a function that also depends on s could lead to the kind of undesirable interference above. Thus through the rare discrete function and universal types, it is possible to have side effects. We will call these types *active* discrete types; discrete types involving only $\star$, $\sqcup$, $\exists$ and base types are *passive*. An obvious but conservative solution to this interference problem is to insist on *passive* discrete types rather than just discrete types in the $\triangleright$To$\star$ and SR rules. The solution is conservative in the sense that passivity is really a property of terms, not types. A typed-term should be declared passive or active only after inspecting the term part for operations that have side-effects (assignment statements, for example). If the typed-term is an assumption, and it is used in a situation requiring a passive object, then it should be tagged with a passivity constraint. If the assumption is later replaced with another term via an elimination rule, the passivity constraint should be propagated to the new term, or to the corresponding component of the new term if more than one assumption is being

[8]The evaluation of such a term can have side effects (the assignment statement, whose type is $\mathbf{1}$, being an example), and this is why all terms but the last have their evaluation forced using **letemp** definitions.

eliminated at once (as in the $\star$E rule). If an active term is substituted for an assumption with a passivity constraint, then the substitution is illegal. Declaring all discrete function types as active is a shortcut that side-steps this delayed legality checking, at the risk of excluding some functions that are in fact discrete.

Unfortunately, excluding all the discrete function types leads to a serious problem. When we develop full-specification for variables and arrays later on, the update rules for arrays will produce effect-statements that include a proposition

$$\forall j\!:\!\mathbf{N}.\, \forall h\!:\! j < n.\ (j < i \sqcup i < j) \multimap (\text{element } j \text{ is unchanged})$$

where i is the index of the updated component and n is the size of the array. The proposition says that all elements other than the ith are unaffected by the update. In the implication the right-hand side is an equality and thus discrete, and the left-hand side is an additive disjunction of inequalities, which are single-atom types. This means that the implication is a discrete type, and it turns out that the corresponding term is simply a passive function on predicate witnesses. To determine the composite effect of a sequence of operations that interfere, their effect-statements must be composed using the $\triangleright$ connective and then transformed using a modified version of the $\triangleright$To$\star$ rule, and there are situations in which the above implications must be composed in just such a way. Our conservative shortcut to identifying active terms makes this composition illegal. Thus a passive-discrete object must be defined as a passive *term* of discrete *type*. We will continue to use the phrase "passive-discrete" in talking about type-term pairs or even types alone, but with the understanding that passivity is a property of the associated term, while discreteness is primarily a property of the type.

Thus our interference control strategy requires deciding in advance which terms interfere, further restricting the $\triangleright$To$\star$ rules to insist on passive discrete types, and permitting no interference across commas in left contexts. This last rule translates into checks on subcontext interference that are outlined in Table 3.4.

Practical Difficulties

There are a couple of problems with this interference control strategy. The first is the cumbersome requirement that which terms interfere must be formally decided in advance. In an implementation of LRCTT that attempted to identify undesirable interference, this would require something equivalent to a declaration of the **ms** set for each assumption. We will see in the last section that through a minor change in the logic this can be dispensed

Comma and Semicolon Rules

Derived Sequent	Restriction
$\Gamma_1, \ldots, \Gamma_n$	$\text{ms}\ (\Gamma_i) \cap \text{ms}\ (\Gamma_j) = \emptyset$ for all $i \neq j$
$\Gamma_1; \ldots; \Gamma_n$	none

Hole-Filling Rules

Type of Hole	Restrictions
$\Delta_l, \bullet$	$\text{ms}\ (\Delta_l) \cap \text{ms}\ (\Gamma) = \emptyset$
$\Delta_l; \bullet$	none
$\bullet, \Delta_r$	$\text{ms}\ (\Gamma) \cap \text{ms}\ (\Delta_r) = \emptyset$
$\bullet; \Delta_r$	none
$\Delta_l, \bullet, \Delta_r$	$\text{ms}\ (\Delta_l) \cap \text{ms}\ (\Gamma) = \emptyset \wedge$ $\text{ms}\ (\Gamma) \cap \text{ms}\ (\Delta_r) = \emptyset$
$\Delta_l; \bullet, \Delta_r$	$\text{ms}\ (\Gamma) \cap \text{ms}\ (\Delta_r) = \emptyset$
$\Delta_l, \bullet; \Delta_r$	$\text{ms}\ (\Delta_l) \cap \text{ms}\ (\Gamma) = \emptyset$
$\Delta_l; \bullet; \Delta_r$	none

Table 3.4: Interference control strategies

with. The second is that applications of the †Et rule to a context created following our rules can produce a context in which interfering terms are separated only by commas. Before presenting an example, we introduce the notion of a threadable context:

Definition 2 A context Γ is *threadable* if every semicolon can be removed through applications of $\star$Eser$_1$ and †Et to produce an independent context in which no two terms interfere. A context which is not threadable is called unthreadable.

A context Γ is *minimally* threadable if it can be transformed to an interference-free independent context by applying only the †Et rule.

In a threadable context all interfering terms must be aliases. Every interference-free independent context is trivially threadable, and every subcontext of a threadable context is itself threadable. A subcontext of a minimally threadable context is threadable but not necessarily minimally threadable.

Context threadability is not necessarily preserved by the inference rules of LRCTT. For example, suppose that

$$\Gamma = a : A,\ b : B,\ c : C$$
$$\Delta = d : A,\ e : B,\ f : C$$

are both interference free and hence threadable, but that a and d, b and e, and c and f are all aliases for the same mutable storage. Then neither

$$\Gamma, \Delta = a : A,\ b : B,\ c : C,\ d : A,\ e : B,\ f : C$$

nor

$$\Gamma; \Delta = a : A,\ b : B,\ c : C;\ d : A,\ e : B,\ f : C$$

is threadable. In the first case, this is obvious, since there are no intervening semicolons. In the second case, there need to be three distinct semicolons, one intervening between each distinct pair of aliases. If we were to permute d, e, f to f, d, e by Ser and then collapse $c; f$ to some $g : C$ by †Et we would remove the one semicolon and $a : A,\ b : B,\ g : C,\ d : A,\ e : B$ would be unthreadable.

Because each application of †Et removes exactly one semicolon, if the only interfering terms in $|\Gamma|$ are aliases then we can count the number of semicolons Γ requires to be minimally threadable. Since interference is aliasing in $|\Gamma|$, the interferes-with relation restricted to Γ is an equivalence relation which partitions $|\Gamma|$ into alias sets. Each of these alias sets represents a *thread*. If Γ is threadable then each thread τ must be collapsible to

a single term via $|\tau| - 1$ applications of †Et. Thus the number of semicolons required is $|\Gamma| -$ (number of threads in Γ). Of course, the $|\tau| - 1$ semicolons for τ must appear between its elements.

We can guarantee that Γ is minimally threadable if we ensure that each term which interferes with any term in the (left) subcontext of Γ to its left has a semicolon on its immediate left. If we wish to ensure that Γ will not lead to unthreadability of any context of which it is made a part, we can do so by ensuring that every nonapplicative term in Γ has a semicolon to its immediate left unless it is the very first term in Γ. Thus nonapplicative terms are combined with others using $\triangleright$I or one of the other rules that introduce semicolons.

3.2 Variables and Arrays

LRCTT allows us to model updatable storage, but the model is unwieldy. In this section we generalize the notion of a storage cell and develop some syntactic sugar and specialized inference rules that represent traditional variables and arrays. Let A be a discrete type. Then we define the *component type set* Δ_A of A as

$$\begin{aligned} \Delta_A &:= \{A\} && \text{if } A \text{ is not a } \star \text{ type} \\ &:= \{A\} \cup \Delta_{A_1} \cup \ldots \cup \Delta_{A_n} && \text{if } A = A_1 \star \ldots \star A_n . \end{aligned}$$

For example, the component type set of $A \star (B \star C) \star (D \sqcup E)$ is

$$\{A \star (B \star C) \star (D \sqcup E),\ A,\ B \star C,\ B,\ C,\ D \sqcup E\}.$$

For a discrete type A the elements of Δ_A are all themselves discrete. We will use the component type set of a discrete type to develop enhanced storage cell types that represent variables and records. First we introduce a small amount of notation. Let S be a finite set of types, A a type, $\circ$ a unary connective, $\odot$ an associative binary connective, and $\ominus$ an arbitrary binary connective; then define

$$\begin{aligned} \circ S &:= \{\circ B : B \in S\} \\ \odot\{B_1, \ldots, B_n\} &:= B_1 \odot \ldots \odot B_n \\ A \ominus S &:= \{A \ominus B : B \in S\} . \end{aligned}$$

Define $A\,\mathbf{var}$, the type of a variable that holds values of type A, as

$$A\,\mathbf{var} := \dagger\,((\sqcap\,\Delta_A)\ \sqcap\ (\sqcap \sim\!\Delta_A))$$

where the first half of the components of the additive conjunction represent fetch operations that retrieve the current value of the parts of the cell, and the second half represent

destructive update operations on the parts of the cell. For example, a variable that holds values of type $\mathbf{N} \star \mathbf{N}$ has type

$$(\mathbf{N} \star \mathbf{N})\,\mathbf{var} \equiv \dagger\,(\mathbf{N} \star \mathbf{N} \ \sqcap\ \mathbf{N} \ \sqcap\ \mathbf{N} \ \sqcap\ {\sim}(\mathbf{N} \star \mathbf{N}) \ \sqcap\ {\sim}\mathbf{N} \ \sqcap\ {\sim}\mathbf{N})$$

where the components of the additive conjunction are

$\mathbf{N} \star \mathbf{N}$, representing the current value of the cell;

$\mathbf{N}$, representing the current value of the first component;

$\mathbf{N}$, representing the current value of the second component;

${\sim}(\mathbf{N} \star \mathbf{N})$, representing a function that updates the whole cell in one shot;

${\sim}\mathbf{N}$, representing a function that updates the first component of the cell only, leaving the second component untouched; and

${\sim}\mathbf{N}$, representing a function that updates the second component of the cell only, leaving the first component untouched.

This new construction can be fully justified using only Reddy's storage cells. Define

$$\begin{array}{lll} A\,\mathbf{var} & := & \dagger\,(A \sqcap {\sim}A) & \text{if } A \text{ is not a } \star \text{ type} \\ A\,\mathbf{var} & := & \dagger(A_1\,\mathbf{var} \sqcap \ldots \sqcap A_n\,\mathbf{var}) & \text{if } A = A_1 \star \ldots \star A_n; \end{array}$$

then we can verify that all of the components of the first definition for $A\,\mathbf{var}$ can be implemented using $\dagger$E and applications of $\sqcap$E$_i$ with this one. Our original definition can be viewed as a sugaring of this one.

We now develop special rules for using an $A\,\mathbf{var}$ that hide its structure. First, associate symbolic type names with each component of the additive conjunction in $A\,\mathbf{var}$ as follows. We number each component of each multiplicative conjunction in A consecutively from 1, and then subscript A with the unique sequence of numbers that identifies the path from the top of A (treated as a tree) to the desired component. For example, letting $A := (\mathbf{N}\star(\mathbf{N}\star\mathbf{N}))$ the component type set Δ_A together with the component names is

$$\{A : \mathbf{N} \star (\mathbf{N} \star \mathbf{N}),\ A_1 : \mathbf{N},\ A_2 : \mathbf{N} \star \mathbf{N},\ A_{2,1} : \mathbf{N},\ A_{2,2} : \mathbf{N}\}.$$

where we write A subscripted with the empty sequence as just A. Now $A\,\mathbf{var} := \dagger((\sqcap\Delta_A)\sqcap(\sqcap{\sim}\Delta_A))$, and by convention we arrange the components Δ_A and ${\sim}\Delta_A$ in lexicographic

order on the identifying sequences. We assign $\mathbf{get}_\nu$ to component A_ν, and $\mathbf{put}_\nu$ to component $\sim A_\nu$. Each of these named components of $A\,\mathbf{var}$ can be picked out by first applying $\dagger$E and then $\sqcap\text{E}_i$ for the desired i. Thus

$$\begin{aligned}\mathbf{get}_\nu(v) &:= \pi_{j_\nu}(\text{RGet}(v)) : A_\nu\\ \mathbf{put}_\nu(v) &:= \pi_{i_\nu}(\text{RGet}(v)) : \sim A_\nu\end{aligned}$$

where A_ν is in position j_ν and $\sim A_\nu$ is in position i_ν. Henceforth we will not explicitly derive $\mathbf{get}_\nu(v) : A$ or $\mathbf{put}_\nu(v) : \sim A_\nu$.

Now, since each A_ν is discrete, we can extract the $\mathbf{put}_\nu$ component and use $\triangleright$I, Sfunc and $\dagger$Et to obtain:

$$\cfrac{\cfrac{v : A\,\mathbf{var} \vdash \mathbf{put}_\nu(v) : \sim A_\nu \qquad \vdash W_1 : \mathbf{1}}{v : A\,\mathbf{var} \vdash [\mathbf{put}_\nu(v)\,;\,W_1] : \sim A_\nu \triangleright \mathbf{1}}}{\begin{array}{l}v : A\,\mathbf{var} \vdash \lambda x.\,\text{let } [f;\,r] = [\mathbf{put}_\nu(v)\,;\,W_1] \text{ in}\\ \qquad (f\,x);\,r : A_\nu \multimap \mathbf{1}\end{array}}$$

We will call the $A_\nu \multimap A\,\mathbf{var}$ term on the right of this sequent $v :=_\nu \_$. Notice that we could also have abstracted out $v : A\,\mathbf{var}$ to obtain a function $:=_\nu : A\,\mathbf{var} \multimap A_\nu \multimap A\,\mathbf{var}$, but we will not need to for the current purposes. Given the sequent

$$v : A\,\mathbf{var} \vdash v :=_\nu \_ : A_\nu \multimap A\,\mathbf{var}$$

and a sequent $\Gamma \vdash v : A\,\mathbf{var}$ of interest we can use $\triangleright$I, SApply and Cut to derive

$$\cfrac{\Gamma \vdash v : A\,\mathbf{var} \qquad \cfrac{\cfrac{\Delta \vdash a : A_\nu \qquad s : A\,\mathbf{var} \vdash s :=_\nu \_ : A_\nu \multimap \mathbf{1}}{\Delta;\,s : A\,\mathbf{var} \vdash [a;\,s :=_\nu \_] : A_\nu \triangleright (A_\nu \multimap \mathbf{1})}}{\begin{array}{l}\Delta;\,s : A\,\mathbf{var} \vdash \text{let } [i;\,f] = [a;\,s :=_\nu \_] \text{ in}\\ \qquad \text{letemp } t_1 = i \text{ in}\\ \qquad (f\,t_1) : \mathbf{1}\end{array}}}{\begin{array}{l}\Delta;\,\Gamma \vdash \text{let } s = v \text{ in}\\ \qquad \text{let } [i;\,f] = [a;\,s :=_\nu \_] \text{ in}\\ \qquad \text{letemp } t_1 = i \text{ in}\\ \qquad (f\,t_1) : \mathbf{1}\end{array}}$$

and we can use $\multimap$E and Cut to derive

$$\cfrac{\Gamma \vdash v : A\,\mathbf{var} \qquad \cfrac{\Delta \vdash a : A_\nu \qquad s : A\,\mathbf{var} \vdash s :=_\nu \_ : A_\nu \multimap \mathbf{1}}{\Delta,\,s : A\,\mathbf{var} \vdash (s :=_\nu a) : \mathbf{1}}}{\Delta,\,\Gamma \vdash \text{let } s = v \text{ in } (s :=_\nu a) : \mathbf{1}.}$$

The two constructions can be viewed as assignment inference rules that abstract away the structure of $A\,\mathbf{var}$, where the first is the sequential version of the second. Compiling away

the unnecessary **let** statements we obtain a pair of inference rules

$$\frac{\Delta \vdash a : A_\nu \qquad \Gamma \vdash v : A\,\mathbf{var}}{\Delta, \Gamma \vdash (v :=_\nu a) : \mathbf{1}}\ (\mathrm{Put}_\nu) \qquad \frac{\Delta \vdash a : A_\nu \qquad \Gamma \vdash v : A\,\mathbf{var}}{\Delta; \Gamma \vdash \begin{array}{l}\mathrm{letemp}\ t_1 = a\ \mathrm{in} \\ v :=_\nu t_1 : \mathbf{1}\end{array}}\ (\mathrm{SPut}_\nu)$$

for each component name ν of A. The sequential rule must be used in situations where the value a depends on v; to ensure threadability it should be used whenever a depends on any nonapplicative term.

In a sense the Put$_\nu$ rule is redundant, because after applying SPut$_\nu$ we can use the Ser rule to change the semicolon between Δ and Γ to a comma:

$$\cfrac{\cfrac{\Delta \vdash a : A_\nu \qquad \Gamma \vdash v : A\,\mathbf{var}}{\Delta; \Gamma \vdash \mathrm{letemp}\ t_1 = a\ \mathrm{in}\ v :=_\nu a\ :\ \mathbf{1}}}{\Delta, \Gamma \vdash \mathrm{letemp}\ t_1 = a\ \mathrm{in}\ v :=_\nu a\ :\ \mathbf{1}.}$$

However, the code generated has an unremovable **letemp** which might prevent some simplification. We will keep both inference rules as a matter of convenience.

For the value component of a $A\,\mathbf{var}$ type the obvious collection of retrieval rules is

$$\frac{\Gamma \vdash s : A\,\mathbf{var}}{\Gamma \vdash \mathbf{get}_\nu(s) : A_\nu}\ (\mathrm{Get}_\nu)$$

Notice that with the Get$_\nu$ rules there is no need for a **var**E elimination rule. The inference rules for $A\,\mathbf{var}$ are rounded out by taking relevant rules for † and specializing them for $A\,\mathbf{var}$. The complete set of inference rules for variables is shown in Table 3.5. There is no rule of the form

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash \mathrm{New}(s) : A\,\mathbf{var}}\ (\mathbf{var}\mathrm{I}) \quad \begin{array}{l}\text{if } \Gamma \text{ is an independent context with} \\ \text{only !, } \mathbf{var}\text{, or } \mathbf{array} \text{ terms}\end{array}$$

corresponding to †I. This is because there is no corresponding introduction rule

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{New}(a) : \dagger(A \sqcap {\sim}A)}\ (\mathrm{cellI}) \quad \begin{array}{l}\text{if } \Gamma \text{ is an independent context with} \\ \text{only !, } \mathbf{var}\text{, or } \mathbf{array} \text{ terms}\end{array}$$

for Reddy's storage cell, and without it **varI** could not be derived. Such a rule would have to be introduced axiomatically; intuitively, if we regard **varI** as creating a new variable and initializing it the rule seems reasonable, but if all the formulas in Γ were banged we could then derive a sequent of the form

$$\Gamma \vdash \mathrm{Store}(v) : !A\,\mathbf{var}.$$

Since a banged object can be copied via contraction elimination, this would allow aliasing of variables. From a programming language perspective this is a potentially useful feature,

$$\frac{\Gamma \vdash s : A\,\mathbf{var}}{\Gamma \vdash \mathbf{get}_\nu(s) \;:\; A_\nu}\ (\mathrm{Get}_\nu)$$

$$\frac{\Delta \vdash s : A_\nu \qquad \Gamma \vdash v : A\,\mathbf{var}}{\Delta, \Gamma \vdash (v :=_\nu a) \;:\; \mathbf{1}}\ (\mathrm{Put}_\nu) \qquad \frac{\Delta \vdash a : A_\nu \qquad \Gamma \vdash v : A\,\mathbf{var}}{\Delta; \Gamma \vdash \begin{array}{l}\mathrm{letemp}\ t_1 = a\ \mathrm{in}\\ v :=_\nu t_1 \;:\; \mathbf{1}\end{array}}\ (\mathrm{SPut}_\nu)$$

$$\frac{\Gamma \vdash s : A\,\mathbf{var}}{\Gamma \vdash \mathrm{Done}(s) \;:\; \mathbf{1}}\ (\mathbf{var}\mathrm{Ew})$$

$$\frac{\Gamma \vdash s : A\,\mathbf{var} \qquad \Delta[x : A\,\mathbf{var};\ y : A\,\mathbf{var}] \vdash t : B}{\Delta[\Gamma] \vdash \mathrm{let}\ x = y = s\ \mathrm{in}\ t \;:\; B}\ (\mathbf{var}\mathrm{Et})$$

Table 3.5: Rules for variables

but as pointed out in section 3.1 from a specification perspective we require control over aliasing.

We now develop a similar treatment for arrays. For any $n \in \mathbf{N} - \{0\}$, define $A\,\mathbf{array}\,[n]$, the type of an n-place array holding values of type A, as

$$A\,\mathbf{array}\,[n] := \dagger\Big(\begin{array}{l} (\sqcap\,((\exists i : \mathbf{N}.\, i < n) \multimap \Delta_A))\ \sqcap \\ (\sqcap \sim((\exists i : \mathbf{N}.\, i < n) \star \Delta_A)) \end{array} \Big)$$

The types $((\exists i : \mathbf{N}.\, i < n) \multimap \Delta_A)$ represent indexed fetch operations on the array members, complete with an index range check. The types $\sim((\exists i : \mathbf{N}.\, i < n) \star \Delta_A)$ represent destructive update functions on the array members, again with index range check. We could similarly define multidimensional arrays $A\,\mathbf{array}[n_1]\ldots[n_k]$ by using multidimensional guard conditions with type

$$(\exists i_1 : i_1 < n_1) \star \ldots \star (\exists i_k : i_k < n_k),$$

but we will not need them for our purposes[9].

Proceeding as with **var** types we can construct a collection of inference rules for use with **array** $[n]$ types that hide the internal structure. Note however an important difference from the **var** rules: rules now have three inputs and so there are four types of *put* rules, each introducing a different combination of commas and semicolons, and two types of *get* rules.

[9]With our approach multidimensional arrays *must* be introduced in this way. Recall from the section on interference control that the fields of updatable objects must be of discrete type and passive. An array type is not discrete because of the $\dagger$ modality (it represents a sequence of array objects), but more important, arrays and variables are not passive. If we could, for example, have an array of arrays, then if we interpret an array as a reference we could copy array references, and then arrays would not be uniquely named and we would undermine our interference control strategy.

$$\frac{\Gamma \vdash a : A\,\mathbf{array}[n] \qquad \Omega \vdash g : (\exists i : \mathbf{N}.\, i < n)}{\Omega; \Gamma \vdash \mathrm{letemp}\, t_1 = g \,\mathrm{in}\, *_\nu\, a\,[t_1] \;:\; A_\nu} \quad (\mathrm{SGet}_\nu)$$

$$\frac{\Omega \vdash g : (\exists i : \mathbf{N}.\, i < n) \qquad \Delta \vdash t : A_\nu \qquad \Gamma \vdash a : A\,\mathbf{array}[n]}{\Omega; \Delta; \Gamma \vdash \begin{array}{l}\mathrm{letemp}\, t_1 = t \,\mathrm{in} \\ \mathrm{letemp}\, t_2 = g \,\mathrm{in} \\ a\,[t_2] :=_\nu t_1 \;:\; \mathbf{1}\end{array}} \quad (\mathrm{SSPut}_\nu)$$

$$\frac{\Gamma \vdash a : A\,\mathbf{array}[n]}{\Gamma \vdash \mathrm{Done}\,(a) \;:\; \mathbf{1}} \quad (\mathbf{array}\,[n]\mathrm{Ew})$$

$$\frac{\Gamma \vdash a : A\,\mathbf{array}[n] \qquad \Delta[x : A\,\mathbf{array}[n]\,;\, y : A\,\mathbf{array}[n]] \vdash t : B}{\Delta[\Gamma] \vdash t : B} \quad (\mathbf{array}\,[n]\mathrm{Et})$$

Table 3.6: Rules for arrays.

We call the *get* rules SGet_ν and Get_ν and the *put* rules SSPut_ν, SIPut_ν, ISPut_ν, and Put_ν, where "I" means "independent". The fully sequential versions and the specialized elimination rules are shown in Table 3.6. As with variables, the fully sequential versions alone would suffice, at the expense of more **letemps**. Again we retain them all for convenience.

3.3 Full Specification for Variables and Arrays

LRCTT, sugared with the above variable and array constructions, forms a typed imperative programming language for which we have an interference control regimen. But this falls short of a system for specification and extraction of imperative programs. Such a system must provide a way to state and reason about the current contents of mutable objects and the effects of destructive updates.

A promising approach is to change the get and put operations so they provide information about the state of the variable. This requires changing the type $A\,\mathbf{var}$ to something like

$$\dagger\,((\exists v : A.\, \mathbf{self} = v) \sqcap (\forall v : A.\, \mathbf{self} = v))$$

assuming a base-type A for simplicity, and where **self** is taken to refer to the object of type $A\,\mathbf{var}$. The existential represents **get**: it fetches the value currently in **self**; the universal represents **put**: it makes **self** equal to any given value of the correct type[10]. The notion of

[10]The put component may seem strangely typed. In FOL, this statement would mean that **self** is simultaneously equal to every $v \in A$. But recall from Chapter 2 that the linear universal is interpreted as a choice of one of many possible alternatives. Once the choice is made, the other alternatives are unavailable.

self is unfortunately problematic because for the same type it must represent many different objects. There is fortunately a way to fix this problem.

Variables

Let A **varname** be a new discrete type, different for each (discrete) type A, and replace A **var** with the new predicate type $A\,\mathbf{var}\,[x : A\,\mathbf{varname}]$ defined as

$$\dagger\Big(\ (\sqcap \{(\exists v : A_\nu . !(*_\nu x = v) \ \star \ \mathbf{same}[*x]) : \nu \in \text{names}(\Delta_A)\} \ ) \ \sqcap \\ (\sqcap \{(\forall v : A_\nu . !(*_\nu x = v) \ \star \ \mathbf{diff}[*_\nu x]) : \nu \in \text{names}(\Delta_A)\} \ ) \Big)$$

where $*_\nu$ is the fetch operator for component ν. A term of type A **varname** is analogous to a pointer, reference, or lvalue, whereas the redefined $A\,\mathbf{var}\,[x]$ can be thought of as the thing pointed at, with accessors for reading and writing the components. The x in $A\,\mathbf{var}\,[x]$ is the canonical reference to the variable; every distinct $x : A\,\mathbf{varname}$ is understood to refer to distinct mutable storage, and two variables $s : A\,\mathbf{var}\,[x]$ and $t : A\,\mathbf{var}\,[y]$ interfere if and only if $x = y$, in which case they are the same variable. The special predicates $\mathbf{same}[*x]$ and $\mathbf{diff}[*_\nu x]$ indicate what change has taken place in the variable as a result of the operation. As the names suggest, $\mathbf{same}[*x]$ means that $*x$ has not changed because of the operation, and $\mathbf{diff}[*_\nu x]$ means that only component ν has changed, or equivalently, that all components of which it is not a part have remained the same. The predicates have expansions

$$\begin{array}{rcll} \mathbf{same}[*x] & := & !(*x = *x') & \\ \mathbf{diff}[*_\nu x] & := & \displaystyle\mathop{\bigstar}_{\nu' \in \mathsf{Sib}(\nu)} !(*_{\nu'} x = *_{\nu'} x') & \text{if Sib}(\nu) \neq \emptyset \\ & := & \mathbf{1} & \text{otherwise} \end{array}$$

where $\text{Sib}(\nu)$ is the set of siblings of ν, that is

$$\text{Sib}(\nu) := \{\rho \cdot \langle a \rangle : \nu = \rho \cdot \langle b \rangle,\ b \neq a\},$$

and we have adopted a priming convention for variable names, where $*x'$ indicates the contents of the storage referenced by x prior to the operation[11]. We supply a set of consistency axioms

$$\vdash\ !(*_\nu x = *_\nu x') \multimap !(*_{\nu'} x = *_{\nu'} x') \quad \text{if } \nu \text{ is a prefix of } \nu'$$

which indicate that if a component of $A\,\mathbf{var}\,[x]$ is unchanged, then so are all components of that component.

[11]This breaks with traditional notations that use primes to denote an object *after* a change. This notation will prove more convenient for us, however, because there are situations where the current value of an object is described in terms of two or more distinct past values. The second last value of $*x$ is $*x''$, the third last $*x'''$, *etc.*

Thus the type A **var** is split into a set of types, each of which is parameterized by its own variable name, so that the get and put components can return information about their effects. A statement of the composite effect of a sequence of operations is obtained by sequencing with $\triangleright$, converting to a $\star$ formula, and then combining individual effects as desired. Using ! types allows effect-information to be copied or discarded if it it not needed. Since ! types are never discrete, to reduce an effect of the form $!E_1 \triangleright E_2$ the first component must be used to generate some finite number of copies $E_1 \star \ldots \star E_1$; if E_1 is discrete then the conjunction is too, and $(E_1 \star \ldots \star E_1) \triangleright E_2$ can itself be reduced to a conjunction. How many copies of E_1 are required must be determined in advance.

The conversion of a sequence of effects $E_1 \triangleright E_2$ to the strict pair $E_1 \star E_2$ requires an adjustment to the $\triangleright$To$\star$ rule. E_1 and E_2 may be predicates over the same variable x, and if E_2 describes a change in x then the conjuncts in $E_1 \star E_2$ may contradict one another. For example, assigning 1 and then 2 to x will result in the composite effect $*x = 1 \ \triangleright \ *x = 2$ and the converted form $*x = 1 \ \star \ *x = 2$ is obviously nonsense. The problem is that $*x = 1$ is what was true *before* $*x = 2$ became true; we reuse the priming convention from the get and put rules to fix this. Any occurrence of a variable that is modified on the right of the $\triangleright$ in the type E_1 is decorated with a prime upon conversion to $\star$ form. The modified $\triangleright$To$\star$ rule is thus

$$\frac{\Gamma \ \vdash \ s : A \triangleright B}{\Gamma \ \vdash \ \text{let } [x; y] = s \text{ in letemp } t_1 = x \text{ in } (t_1, y) \ : \ A' \star B} \quad (\triangleright\text{To}\star)$$

$$\begin{array}{l} \text{where } A' \ = \ A[x_1'/x_1, \ldots, x_n'/x_n] \\ \text{for each } x_i \text{ that depends on some} \\ \text{variable or array appearing in } B \end{array}$$

where "mutable names" refers to variable names and array names, which are introduced shortly.

There is an immediate problem with our use of A **varnames**: any two terms that are both of type A **var** $[x]$ can modify the same variable, so aliasing has been introduced. We correct this by forbidding identical **var** types in the same independent context. Thus

$$a : A \ \textbf{var} \ [x] \, ; \, b : A \ \textbf{var} \ [x]$$

may appear on the left side of a turnstile, but

$$a : A \ \textbf{var} \ [x] \, , \, b : A \ \textbf{var} \ [x]$$

may not. We can also fix this problem by making the **varnames** intrinsically regenerative and providing a function $\textbf{access}_A$ for each (passive) discrete type A that for any

$x : A\,\mathbf{varname}$ returns a $A\,\mathbf{var}\,[x]$, and forbid the introduction of any $A\,\mathbf{var}\,[x]$ by assumption. Thus every $A\,\mathbf{var}\,[x]$ will have $x : A\,\mathbf{varname}$ in its left context. By our restrictions on context combinations the same $A\,\mathbf{varname}$ cannot appear twice in any independent context, so neither could distinct $v_1 : A\,\mathbf{var}\,[x]$ and $v_2 : A\,\mathbf{var}\,[x]$. Either approach is workable and requires additional restrictions; for the developments in this chapter the first approach simplifies the discussion.

As before, the get and put components can be projected out by using †E and the appropriate ⊓E rule; we rename the terms

$$\begin{aligned} *_\nu s &:= \pi_{i_\nu}\,(\mathrm{RGet}\,(s)) \\ (s :=_\nu \_) &:= \pi_{j_\nu}\,(\mathrm{RGet}\,(s)) \end{aligned}$$

where i_ν and j_ν are the positions of the get and put components in the conjunction, respectively. We will not show the derivations of these components in examples.

The only change in the rules for variables are in Get and (S)Put; the new versions are shown in Table 3.7. The final term for Get_ν is an existential pair comprised of the current

$$\frac{\Gamma \vdash s : A\,\mathbf{var}\,[x]}{\Gamma \vdash (\,*_\nu s, W_s\,(*_\nu s, *_\nu x)\,) \;:\; \exists v : A_\nu.\,!(*_\nu x = v) \star \mathbf{same}\,[*x]}\;(\mathrm{Get}_\nu)$$

$$\frac{\Delta \vdash v : A_\nu \qquad \Gamma \vdash s : A\,\mathbf{var}\,[x]}{\begin{array}{ll}\Delta;\Gamma \vdash & \mathrm{letemp}\; t_1 = s :=_\nu v \;\mathrm{in} \\ & t_1 \;:\; !(*_\nu x = v) \star \mathbf{diff}\,[*_\nu x]\end{array}}\;(\mathrm{SPut}_\nu)$$

Table 3.7: New Get_ν and SPut_ν rules for variables.

value, named $*_\nu s$ instead of $*_\nu x$, and a witness of the equivalence of $*_\nu s$ and $*_\nu x$. The term for SPut_ν is a lazy sequence containing an update of s and a new reference to s.

Incrementing Revisited

To illustrate effect-composition, we will derive an increment operation for a variable of type $\mathbf{N}\,\mathbf{var}\,[x]$. Rather than present one large incomprehensible sequent calculus derivation, we show important subderivations and describe how they can be composed to get the final product. We also suppress the **same** and **diff** predicates in this derivation, since they are unnecessary for illustrating the key points.

The first step is to read the variable:

$$\frac{v : \mathbf{N}\,\mathbf{var}\,[x] \vdash v : \mathbf{N}\,\mathbf{var}\,[x]}{v : \mathbf{N}\,\mathbf{var}\,[x] \vdash (*v, p) \;:\; \exists y : \mathbf{N}.\,!(*x = y)}$$

The sequent

$$y_1 : \mathbf{N},\ p_1 : !(*x = y_1) \ \vdash\ (y_1 + 1, W(p_1)) : \exists z : \mathbf{N}. !(z = *x + 1)$$

is straightforward to derive and can be used to eliminate the existential above, obtaining

$$\begin{array}{ll} v : \mathbf{N}\,\mathbf{var}\,[x] \vdash & \text{let } (y_1, p_1) = (*v, p) \text{ in} \\ & (y_1 + 1, W(p_1)) : \exists z : \mathbf{N}.!(z = *x + 1), \end{array}$$

where $W(p_1)$ is a complex witness term containing p_1, which we suppress for clarity. We would like to consume this existential pair by updating the variable x. We can show that the existing SSPut rule together with rules for manipulating $\star$ and $\triangleright$ expressions and $\exists$E derive

$$\frac{\Delta \ \vdash\ \epsilon : \exists y : A_\nu. P[y] \qquad \Gamma \ \vdash\ v : A\,\mathbf{var}\,[x]}{\begin{array}{ll} \Delta; \Gamma \vdash & \text{let } (a, s) = \epsilon \text{ in} \\ & \text{letemp } t_1 = (a, s) \text{ in} \\ & \text{let } (t_2, t_3) = t_1 \text{ in} \\ & (t_2, (t_3, v :=_\nu t_2)) \ :\ \exists a : A_\nu.\ (P[a]' \star !(*_\nu x = a) \star \mathbf{diff}[*_\nu x]) \end{array}}$$

where $P[a]'$ is the primed version of $P[a]$ obtained by applying $\triangleright$To$\star$. Combining this with the last existential and a fresh[12] $w{:}\mathbf{N}\,\mathbf{var}\,[x]$ we obtain

$$\begin{array}{l} v{:}\mathbf{N}\,\mathbf{var}\,[x];\ w{:}\mathbf{N}\,\mathbf{var}\,[x] \quad \vdash \\ \quad \text{let } (a, s) = \\ \quad\quad \text{let } (y_1, p_1) = (*v, p) \text{ in} \\ \quad\quad (y_1 + 1, W(p_1)) \text{ in} \\ \quad \text{letemp } t_1 = (a, s) \text{ in} \\ \quad \text{let } (t_2, t_3) = t_1 \text{ in} \\ \quad (t_2, (t_3, v := t_2)) \ :\ \exists a : \mathbf{N}.\ !(a = *x' + 1) \star !(*x = a) \star \mathbf{diff}[*x] \end{array}$$

Clearly, the conjuncts within the existential type can be reduced to

$$!(*x = *x' + 1) \star \mathbf{diff}[*x]$$

which is free of a, so by yet another existential elimination and an application of †Et we obtain

$$s : \mathbf{N}\,\mathbf{var}\,[x] \vdash T(s) \ :\ !(*x = *x' + 1) \star \mathbf{diff}[*x]$$

where we have abbreviated the resulting term $T(s)$. The $s : \mathbf{N}\,\mathbf{var}\,[x]$ on the left can be removed by forming a function

$$\vdash\ (\lambda s : \mathbf{N}\,\mathbf{var}\,[x].\, T(s)) \ :\ (\mathbf{N}\,\mathbf{var}\,[x] \multimap\ !(*x = *x' + 1) \star \mathbf{diff}[*x])$$

[12]Remember this is extended LCTT, which is still linear. Hence if we want to use a $\mathbf{N}\,\mathbf{var}\,[x]$ twice we have to introduce dummies and then use the thread rule †Et to collapse them together.

or simply

$$\vdash\ \mathbf{inc}_x\ :\ (\mathbf{N}\,\mathbf{var}\,[x]\ \multimap\ !(*x = *x' + 1)\ \star\ \mathbf{diff}[*x]).$$

This function can only be used to increment the variable named x, but no variable of type $A\,\mathbf{var}\,[y]$ where $y \neq x$, and so isn't tremendously useful. But recall that $A\,\mathbf{varname}$ is a type, so x is a term. Introducing it as an unknown, we could create an object of universally-quantified type that gives us the collection of increment functions for all variables of type $\mathbf{N}$. Moreover, the left-side of the resulting sequent would be empty, so !I could be applied to obtain

$$\text{Store}\,(\boldsymbol{\lambda} x.\,\mathbf{inc}_x)\ :\ (!\,\forall x : \mathbf{N}\,\mathbf{varname}.\ (\mathbf{N}\,\mathbf{var}\,[x]\ \multimap\ !(*x = *x' + 1)\ \star\ \mathbf{diff}[*x]))\,.$$

Applying $\mathbf{inc}_x$ twice in succession to $v : \mathbf{N}\,\mathbf{var}\,[x]$ is simple. Apply it separately to two fresh $v_1 : \mathbf{N}\,\mathbf{var}\,[x]$ and $v_2 : \mathbf{N}\,\mathbf{var}\,[x]$, combine the results using $\triangleright$I, then use †Et to merge the aliases v_1 and v_2 as v. This leaves

$$\begin{array}{l} v : \mathbf{N}\,\mathbf{var}\,[x]\ \ \vdash \\ \quad \text{let } v_1 = v_2 = v \text{ in} \\ \quad [\mathbf{inc}_x\,v_1; \mathbf{inc}_x\,v_2]\ : \\ \qquad (!\,(*x = *x' + 1)\ \star\ \mathbf{diff}[*x]) \\ \qquad \triangleright \\ \qquad (!\,(*x = *x' + 1)\ \star\ \mathbf{diff}[*x]) \end{array}$$

and using a combination of !E, $\triangleright$To$\star$ and !I we can transform the type to

$$!\,(((*x' = *x'' + 1)\ \star\ \mathbf{diff}[*x'])\ \star\ ((*x = *x' + 1)\ \star\ \mathbf{diff}[*x]))$$

which can (provided there are sufficient capabilities for reasoning about natural numbers) be reduced to

$$!(*x = *x'' + 2).$$

In order to prevent long sequences of primes we allow a "compaction" of them, so the above formula may be stated as

$$!(*x = *x' + 1).$$

Arrays

The strategy for changing the definition of arrays is similar. We again propose a new discrete base type $A\mathbf{arrayname}$ for each discrete type A, change $A\,\mathbf{array}\,[n : \mathbf{N}]$ to

$$A\mathbf{array}\,[x : A\mathbf{arrayname}, n : \mathbf{N}]\,.$$

This type has components analogous to those for $A\,\textbf{array}[n:\mathbf{N}]$, except the types are appropriate for arrays. The get function for component ν of the array elements has type

$$\forall i:\mathbf{N}.\ \forall g:i<n.\ \exists v:A_\nu.\ (!(*_\nu x[i,g]=v)\ \star\ \textbf{asame}[x])$$

where the predicate $\textbf{asame}[x]$ expands to

$$!\,(\forall j:\mathbf{N}.\ \forall h:j<n.\ \textbf{same}[*x[j,h]]\,);$$

the put function has type

$$\begin{array}{l}\forall i:\mathbf{N}.\ \forall g:i<n.\ \forall v:A_\nu.\\ \qquad !(*_\nu x[i,g]=v)\ \star\ \textbf{diff}[*_\nu x[i,g]]\ \star\ \textbf{adiff}[x,i,g]\end{array}$$

where $\textbf{adiff}[x,i,g]$ expands to

$$\begin{array}{l}!\,\forall j:\mathbf{N}.\ \forall h:j<n.\\ \qquad (i<j\ \sqcup\ j<i)\multimap\textbf{same}[*x[j,h]].\end{array}$$

$\textbf{asame}[x]$ means that every element of x after the operation is the same as it was before; $\textbf{adiff}[x,i,g]$ means that every element other than the ith is the same, and the ith one is different. Because we must manipulate both the indices and the proofs that they are in range in the **asame** and **adiff** predicates, we have changed the original $g:(\exists i:\mathbf{N}.\,i<n)$ input type of the $A\,\textbf{array}[n:\mathbf{N}]$ get and put functions to $\forall i:\mathbf{N}.\forall g:i<n$, and the array dereference operator now takes an explicit pair of arguments. Again we use ! types to allow repeated use of the effects.

As with variables, the essential differences between $A\textbf{array}[x,n]$ and $A\,\textbf{array}[n]$ are in the Get and (SS)Put rules. The new rules are shown in Table 3.8. The term W is the witness of $!(*_\nu x[i,g]=v)\star\textbf{asame}[x]$ and includes a witness of the equivalence of $*_\nu a[i,g]$ and $*_\nu x[i,g]$ and a witness for $\textbf{asame}[x]$. The witnesses for $\textbf{asame}[x]$ and $\textbf{adiff}[x,i,g]$ will in turn have elaborate forms that reflect their types. The universal types in **asame** and **adiff** are non-discrete and we must develop a procedure for simplifying composite effects that contain them. Recall that the semantics of a universally-quantified type is akin to a giant "your choice." The $\triangleright$ connective distributes over $\sqcap$ on the left; that is, the following (termless) sequent is valid:

$$\vdash\ (A\sqcap B)\triangleright C\ \multimap\ (A\triangleright C)\sqcap(B\triangleright C).$$

This suggests that $\triangleright$ might commute with universal quantification on the left, or that the following sequent is valid:

$$\vdash\ (\forall x:A.\ P[x])\triangleright C\ \multimap\ \forall x:A.\ (P[x]\triangleright C).$$

Indeed, this is the case, and we can consequently justify adding the rule

$$\frac{\Gamma \vdash c : (\forall x : A.\, P[x]) \triangleright C}{\Gamma \vdash (\boldsymbol{\lambda} x.\, \text{let } [F; b] = c \text{ in } [(F\, x); b]) \;:\; \forall x : A.\, P[x] \triangleright B}$$

to our type theory. We can also easily show that

$$\frac{\Gamma \vdash d : (A \star \forall y : B.\, Q[y])}{\Gamma \vdash (\boldsymbol{\lambda} y.\, \text{let } (a, F) = d \text{ in } (a, (F\, y))) \;:\; \forall y : B.\, A \star Q[y]}.$$

If in addition the type $P[x]$ is discrete, or can be reduced to a discrete form, for each $x \in A$, then we can combine the first rule, the $\triangleright$To$\star$ rule, and the second rule to obtain

$$\frac{\Gamma \vdash s \;:\; (\forall x : A.\, P[x]) \triangleright (\forall y : B.\, Q[y])}{\Gamma \vdash r \;:\; \forall x : A.\, \forall y : B.\, P[x]' \star Q[y]}$$

$$P[x] \text{ discrete for all } x \in A$$

where $P[x]'$ is the modified $P[x]$ produced by $\triangleright$To$\star$. The resulting term r is

```
λx.λy.  let (a, G) =
              let [t_1; t_2] = let [F; b] = s in [(F x); b] in
              letemp x_1 = t_1 in
              letemp x_2 = t_2 in
              (x_1, x_2)
        in (a, (G y))
```

which is computationally equivalent to

```
let [F; b] = s in
    λx.λy.
        letemp x_1 = (F x) in
        letemp x_2 = b in
        (x_1, (x_2 y)).
```

We can extend this to conjunctions $E_1 \star \ldots \star E_n$ of universal types by moving quantifiers (renaming bound variables if necessary), eliminating them on fresh unknowns, applying $\triangleright$To$\star$ and then restoring the quantifiers through $\forall$I. The procedure for reducing a composite effect $E_1 \triangleright E_2$ can be summarized as

1. convert E_1 to a discrete form D_1 by

   (a) changing each $!F$ type to $\mathbf{1}$, F, or a finite conjunction $F \star \ldots \star F$; and

   (b) eliminating universals on fresh unknowns (or on values that will be of interest);

2. apply $\triangleright$To$\star$ to obtain $D_1' \star E_2$; and

3. manipulate this type to get the desired final effect.

Since all effects originate with the variable and array get and put rules and these can always be put in discrete form, the procedure will work for arbitrary derivations. An elaborate example is presented in Chapter 5.

$$\frac{\Omega \vdash i : \mathbf{N} \qquad \Lambda \vdash g : i < n \qquad \Gamma \vdash a : \mathbf{Aarray}[x, n]}{\begin{array}{ll} \Omega; \Lambda; \Gamma \vdash & \text{letemp } t_1 = i \text{ in} \\ & \text{letemp } t_2 = g \text{ in} \\ & (\ *_\nu a[t_1, t_2], W\ ) : \\ & \quad \exists v : A_\nu .\, !(*_\nu x[i, g] = v) \star \mathbf{asame}[x] \end{array}} \qquad (\mathrm{SGet}_\nu)$$

$$\text{where } W := (\ W_s(\ *_\nu a[i, g], *_\nu x[i, g]), W_{\mathbf{asame}}(x, i, g)\ )$$

$$\frac{\Omega \vdash i : \mathbf{N} \qquad \Lambda \vdash g : i < n \qquad \Delta \vdash v : A_\nu \qquad \Gamma \vdash a : \mathbf{Aarray}[x, n]}{\begin{array}{ll} \Omega; \Lambda; \Delta; \Gamma \vdash & \text{letemp } t_1 = i \text{ in} \\ & \text{letemp } t_2 = g \text{ in} \\ & \text{letemp } t_3 = v \text{ in} \\ & (\ a[t_1, t_2] :=_\nu t_3, W_{\mathbf{adiff}}(x, i, g)\ ) : \\ & \quad !(*_\nu x[i, g] = v) \star \mathbf{diff}[*_\nu x[i, g]] \star \mathbf{adiff}[x, i, g] \end{array}} \qquad (\mathrm{SSPut}_\nu)$$

Table 3.8: New Get$_\nu$ and SSPut$_\nu$ rules for arrays.

3.4 ICTT: A Convenient LRCTT

Recall that in Chapter 2 we ended by developing a more convenient LCTT called SLCTT that required left contexts with banged intersection instead of empty intersection. The revised logic did away with the nuisance of having several identifiers for objects of banged type. The !Ec rule was no longer necessary to propagate copies of banged objects, but was kept around to allow explicit copying or aliasing if it was desired. We can incorporate this enhancement into LRCTT, and provide a similar enhancement that avoids the forced aliasing of † terms and simplifies the interference control strategy outlined in Section 3.1.

The modification to the logic is simple: we do away with the †Et rule altogether, and allow the same † object to appear several times in a context, provided consecutive instances are separated by a semicolon. In a manner similar to the proof at the end of Chapter 2 that every derivation in SLCTT could be performed in LCTT, we can show that every derivation in this modified LRCTT has an equivalent derivation in LRCTT. By combining the two modifications we obtain ICTT, an imperative constructive type theory. The requirements on argument contexts in the inference rules are summarized in Table 3.9. Notice that for the hole-filling rules the requirements depend on the punctuation around the hole.

Unlike in SLCTT where we retained the !Ec rule, we do not retain †Et. Recall that we restricted LRCTT so that no ! type is nonapplicative. Since nonapplicative types interfere with nothing, we can allow unrestricted aliasing for them; but if we do away with †Et then we can adopt the convention that distinct variable and array names in the same scope

level represent distinct areas of storage. We call a derivation of $\Gamma \vdash s : A$ *complete* if every nonapplicative term in $|\Gamma|$ has a storage type. In such a Γ two nonapplicative terms interfere if and only if they have the same name. If there was a violation of the interference rules in Table 3.4 at some point in the derivation then Γ (or some earlier context) will contain the same nonapplicative term in an independent subcontext, and this will be caught by the checks in Table 3.9, or by attempting to thread the context. This means that in a complete derivation we can do away with declaring the **ms** set for each assumption, perhaps at the expense of not knowing at the earliest possible moment in a derivation that there is illegal interference. Ideally, complete programs require complete derivations, but such an approach disallows using nonapplicative library routines which were not constructively developed. If the system is used to develop such programs some declaration of the storage affected by nonapplicative assumptions corresponding to the library routines is required[13]. The system would simply have to take it on faith that the declarations were correct.

In the next two chapters, we describe an implementation of ICTT and work through an extended example using it.

[13]For example, an assumption introduction rule that allows an arbitrary left context.

Comma and Semicolon Rules

Result context	Restriction on argument contexts
$\Gamma_1, \ldots, \Gamma_n$	**Banged** $[\lvert\Gamma_i\rvert \cap \lvert\Gamma_j\rvert]$ for all $i \neq j$
$\Gamma_1; \ldots; \Gamma_n$	**BangedOrDaggered** $[\lvert\Gamma_i\rvert \cap \lvert\Gamma_j\rvert]$ for all $i \neq j$

Hole-filling rules

Result context	Restriction on argument contexts
$\Delta_l, \bullet$	**Banged** $[\lvert\Delta_l\rvert \cap \lvert\Gamma\rvert]$
$\Delta_l; \bullet$	**BangedOrDaggered** $[\lvert\Delta_l\rvert \cap \lvert\Gamma\rvert]$
$\bullet, \Delta_r$	**Banged** $[\lvert\Gamma\rvert \cap \lvert\Delta_r\rvert]$
$\bullet; \Delta_r$	**BangedOrDaggered** $[\lvert\Gamma\rvert \cap \lvert\Delta_r\rvert]$
$\Delta_l, \bullet, \Delta_r$	**Banged** $[\lvert\Delta_l\rvert \cap \lvert\Gamma\rvert] \wedge$ **Banged** $[\lvert\Gamma\rvert \cap \lvert\Delta_r\rvert]$
$\Delta_l; \bullet, \Delta_r$	**BangedOrDaggered** $[\lvert\Delta_l\rvert \cap \lvert\Gamma\rvert] \wedge$ **Banged** $[\lvert\Gamma\rvert \cap \lvert\Delta_r\rvert]$
$\Delta_l, \bullet; \Delta_r$	**Banged** $[\lvert\Delta_l\rvert \cap \lvert\Gamma\rvert] \wedge$ **BangedOrDaggered** $[\lvert\Gamma\rvert \cap \lvert\Delta_r\rvert]$
$\Delta_l; \bullet; \Delta_r$	**BangedOrDaggered** $[\lvert\Delta_l\rvert \cap \lvert\Gamma\rvert] \wedge$ **BangedOrDaggered** $[\lvert\Gamma\rvert \cap \lvert\Delta_r\rvert]$

Table 3.9: ICTT: restrictions on argument context intersections

Chapter 4

MIZAR-I: Syntax and Implementation

This chapter describes MIZAR-I, a simplified version of ICTT suitable for processing by a computer program, and `mic`, the MIZAR-I proof checker and editor. The important parts of the syntax of the input language are briefly described in Section 4.1. Section 4.2 describes the general structure of MIZAR-I inference rules and presents a couple of examples. Section 4.3 discusses the proof checker software, describing its operation, the features it provides, and the features it lacks. Chapter 5 will describe a small example derivation illustrating how `mic` is used, and the difficulties inherent in the MIZAR-I approach to deriving programs.

In this and the next chapter, all MIZAR-I syntax is written in typewriter font, including types, terms and sequents which have direct counterparts to constructions from Chapter 3. This is to distinguish MIZAR-I objects, which we will view as concrete, from the abstract ideas presented previously.

4.1 Input Languages

MIZAR-I is a collection of input languages based on the developments of Chapter 3. The two main languages are the derivation language and the rules language. The first is used for writing proofs in the system, and the second is used for describing inference rules. The derivation language is described below and the rules language in the next section.

MIZAR-I derivations are structured as modules, one module per file. Module or derivation files should be given names ending in `.mi`. A module has a header beginning with the declaration

```
module <name>;
```

which gives the module a name. By convention, the module name should be the same as the root of the filename, so that module `BigProof` is in file `BigProof.mi`. The module declaration is followed by an `export` command which lists the identifiers that are visible outside the module, and an `import` command which lists modules whose exported identifiers are used in the derivation. The derivation itself starts with the keyword `begin` and runs until the end of the file.

A derivation is a sequence of commands, each of which is terminated with a semicolon and has one of the forms:

`assume <id>:<formula>` : An assumption introduction. This creates a trivial sequent of the form

```
<id>:<formula> |- <id>:<formula>
```

The identifier represents both the resultant term on the right side of the sequent and the sequent itself. This double meaning of the introduced identifiers is typical of the other commands.

`ax <id>:<type>` : An axiom introduction. This is similar to the `assume` command but the new sequent has the form

```
|- <id>:<formula>
```

This command provides a mechanism for naming objects from outside the MIZAR-I system. Built-in constants like the successor function for natural numbers are introduced in this way. Terms that are introduced axiomatically must have external implementations, in a runtime library for instance.

`<id> by <proc> ( <arguments> )` : An inference procedure invocation. The identifier names the new sequent, and can also serve as an alias for the constructed term. The `<proc>` is the inference procedure, which is either the name of an inference rule, or a sequence of names of inference rules separated by commas and enclosed in curly braces. Each argument may by one of

- `<id>` : A sequent identifier.
- `<term>:<formula>` : A term with its type.
- `term <term>` : A term by itself. The keyword `term` is used to distinguish terms that are just identifiers from sequent identifiers.

`type <formula>` : A formula or type by itself.

`num <number>` : A natural number constant. This is a raw number, and is not interpreted as a natural number term.

`<id>: now` : Open a new nested block with the given name.

`end` : Close the last block opened with a `now` command.

Inference procedures operate using a stack discipline. The arguments in the parentheses are pushed onto a stack in right-to-left order, so that the order in which they appear in the command is the order in which they are popped from the stack. The rules that make up the inference procedure are then executed in succession. Each rule pops zero or more arguments from the stack, computes some result which must always be a sequent or a collection of sequents, and pushes the result onto the stack. When the last procedure has finished, the top of the stack is examined. If it is a single sequent, it is popped and saved under the given identifier, becoming part of the body of sequents available for further inference procedure invocations. If it is a collection of sequents, a message is displayed saying that the result of the procedure is ambiguous, listing the choices, and suggesting that the procedure be modified to produce a unique result. Inference procedures are either built-in, or are specified in one or more separate files using the rules language. There are currently only five built-in inference procedures:

`assumption` : the procedure form of the `assume` command;

`axiom` : the procedure form of the `ax` command; and

`select` : a procedure which takes a collection of sequents and a natural number n as arguments, and selects the nth sequent in the collection as the result. This is typically appended to a procedure that has produced an ambiguous result to select one of the alternatives.

`contraction` : a procedure which takes a sequent as argument and produces the same sequent but with repeated identifiers of banged type merged in all independent subcontexts. Thus $a\,{:}!A,\ b\,{:}!B,\ a\,{:}!A$ becomes $a\,{:}!A,\ b\,{:}!B$, but $a\,{:}!A,\ b\,{:}!B;\ a\,{:}!A$ remains the same.

`threading` : a procedure which takes a sequent as argument and produces the same sequent but contracted and with as many semicolons removed as possible through

threading or through !-serialization. A semicolon will be removed if the independent contexts to its immediate left and right both contain the same daggered or banged object, where daggered objects include variable and array names. Preference is given to matching daggered objects, but banged objects will be used if they are available. The procedure will fail if it discovers a repeated object of plain type in any context, or a repeated object of daggered type in an independent subcontext.

Inference procedures may produce ambiguous results because a sequent unification may discover more than one unifier for a given sequent pattern and input, leading to more than one correct result. This is described further in the following sections.

Blocks represent nested scopes within the derivation and improve the readability of proofs. Assumptions, axioms, and derived sequents that are introduced within the block are not visible outside it. A correctly finished block always has a result sequent, and this sequent is bound to the block name when the matching `end` command is read. The block name also identifies the block for the purposes of interactive proof editing. The result sequent is always the last sequent derived in the block, and its left context must not contain any of the local assumptions. Local assumptions can be removed by eliminating them with assumptions from an outside scope, abstracting them to function arguments or quantified objects, or by abstracting them as local variables. Local variable abstraction is like function abstraction but generates a local variable declaration instead of a lambda expression. The local variable is not accessible from outside the resulting term.

Formulas and terms have a syntax which is suggested by that in Chapters 2 and 3, but using sequences of regular typewriter characters in place of special symbols like the quantifiers and the connectives. The syntax is similar enough that there should be no problem reading the examples in Chapter 5 without having to consult a formal grammar[1].

Derivation files include several kinds of comments. C++-style comments are simply ignored. Multi-line comments enclosed in `<*` and `*>` represent documentation for the module, for individual sequents produced through inference procedures, and for blocks. The MIZAR-I system also provides a limited collection of proof editing commands for the interactive portion of the proof checker.

[1]The formal grammars can be made available upon request. They have been left out of the appendicies in the interest of brevity.

4.2 Inference Rules

Inference rules in MIZAR-I are largely specified outside the proof checker. This allows easy modification and debugging of rules, and the development of special rules to speed up derivations that are performed over and over. This is a poor substitute for automated theorem proving, but could be a valuable feature even in a more sophisticated system.

All rule specifications have the same basic structure:

```
<id> : "long name" <* documentation *>
        given   <input patterns>
        construct
                <output pattern>
        [provided <proviso>]
;
```

The identifier gives the rule its name, the long name is intended to be the user-friendly version, and the special comment is saved as documentation for the rule. Rule specifications are declarative in that they indicate what the shape of the input must be through input patterns, and then express the shape of the output in terms of *metavariables* appearing in the input patterns. The optional proviso section contains assertions about properties of the arguments, and cause the rule application to fail if they are not met.

The different types of input patterns correspond closely to the concrete inputs supplied in an inference procedure command. They include

patterns for terms: these are terms augmented with metavariable symbols of the form `?x`, for `x` an identifier;

patterns for types: these are formulas augmented with metavariable symbols of the form `‘A`, for `A` an identifier; they also include parameterized metavariables of the form `‘A[<t1>,...,<tn>]` that assert that the matching input contains the indicated terms[2];

patterns for typed terms: these are pairings of term and type patterns separated with a colon;

patterns for left contexts: these include

raw unknowns: any identifier, but usually single capital letters such as `G` or `D`

[2]The terms are restricted in practice to either parameters or metavariables; searching for an arbitrary subexpression in a term is possible but expensive and we have chosen to avoid it.

unknowns with holes: the holes are raw sequent expressions using typed term patterns that are enclosed in square brackets. For example `D,[?x:‘A]` is a right hole containing a term matching `?x:‘A` and separated from the rest of the context by a comma, unless it is the only term in the context. Left holes are similar but the hole expression and connective appear on the left. An arbitrary hole `D[?x:‘A;?y:‘B]` indicates a context with some subcontext matching the hole pattern; and

patterns for sequents: a pairing of a context pattern and a typed term pattern, separated by a turnstile `|-`.

Input pattern matching is a unification of input patterns with the actual inputs whose output is a set of metavariable bindings. Unification is simple for terms and formulas, but considerably more complex for contexts. Contexts are partially-ordered multisets, and elements separated by commas may be freely permuted without changing the overall meaning, so one pattern can match one input in many ways. For example, the hole pattern `D[?x:‘A,?y:‘B]` must match any context which has an independent subcontext with at least two elements, and there will be a separate match to consider for every possible pair (including permutations of order) of elements in every such subcontext. Even a more specific hole pattern such as `G[x:N,y:Bool]` must match any context in which `x:N` and `y:Bool` appear in the same independent subcontext, whether they appear next to one another, separated by several other terms, or in the order opposite to that in pattern. Thus context unification can be expensive in both time and space, but this is necessary if inference rules that fill holes are to be expressed in a straightforward way. Rules often require extra structural arguments that help cut down the number of possible matches for their context patterns.

Output patterns are simpler: they are always sequent constructions of the form

```
<cc-pattern> |- <term pattern> : <formula pattern>
```

where the `cc-pattern` includes only context variables, connective expressions such as `D,G;K` and hole-filling expressions such as `D[G]` or `[G];D`. The metavariables appearing in the context patterns must occur in the input patterns so that they will be bound to parts of the arguments during input pattern unification.

Provisos allow rules to perform validity checks that are difficult or impossible to express through pattern expressions alone. Provisos are boolean expressions together with a set of

built-in boolean-valued test functions that operate over output patterns. They are evaluated after the input pattern matching has produced bindings for the metavariables. The built-in functions include

`Param(<term>)` : test whether a term is a parameter (lone identifier) and not a more complex construction;

`NatConst(<term>)` : test whether a term is a natural number constant;

`BoolConst(<term>)` : test whether a term is a boolean constant;

`Discrete(<type>)` : test whether a type is discrete;

`PassiveDiscrete(<type>)` : test whether a type is passive and discrete;

`Equals(<type1>,<type2>)` : test whether two types are equal; the formulas may differ only on bound parameters;

`Free(<term>,<type>)` : test whether a term (which must be a parameter) appears free in a type;

`Free(<term>,<context>)` : test whether a term appears free in any of the types appearing in a context; evaluates to *false* if the context is empty;

`Banged(<context>)` : test whether every type appearing in a context is banged; evaluates to *true* if the context is empty; and

`BangedOrDaggered(<context>)` : test whether every type appearing in a context is either banged or daggered; variable and array names are considered intrinsically daggered; evaluates to *true* if the context is empty.

The steps of inference rule evaluation are to match input patterns, check provisos, and then construct the output. An evaluation will fail if the inputs cannot be completely matched, the proviso evaluates to false, or if a proviso or output pattern encounters a metavariable with no binding.

To illustrate we will consider two of the current crop of MIZAR-I inference rules. Multiplicative conjunction introduction is among the simpler rules:

```
rule mci: "* Intro"

        given   G |- ?x:‘A,
```

```
                D |- ?y:‘B
        construct
                G,D |- (?x,?y):‘A*‘B
;
```

The interpretation is straightforward. The two input sequent patterns will each match any sequent (any context with any typed term). If there are indeed two sequents on the stack, they are popped and bindings are immediately found for the metavariables. The output pattern says what the shape of the resulting sequent should be: the first and second argument contexts are glued together with a comma, the terms are assembled into a strict pair, and the types are combined into a multiplicative conjunction.

A more complex rule is equality elimination:

```
rule eqe: "= Elim"


        given   G |- ?s: ?a=?b,
                term ?x,
                type ‘Skel[?x],
                D |- ?t: ‘A
        construct
                G,D |- WEqE(?a,?b,?t): (‘Skel[?x], ?x->?b)
        provided
                Param(?x) and
                not Free (?x, ‘A) and
                Equals (‘A, (‘Skel[?x], ?x->?a))
;
```

This rule replaces some collection of occurrences of ?a in the type ‘A with its equivalent ?b. The term ?x and type ‘Skel[?x] allow the selection of which instances of ?a will be replaced. The provisos insist that ?x is a parameter not already in ‘A such that ‘A and ‘Skel with ?x replaced with ?a are the same. That is, Skel[?x] is a sort of skeleton of ‘A that shows where to plug in copies of ?b. The result type is built by plugging ?b into ‘Skel, and the result term is a witness for the substitution of ?b for a in the type of term ?t.

4.3 The `mic` and `TkMic` Programs

Software for MIZAR-I includes a command-line proof checker called `mic` ("MIZAR-I compiler") with a very simple interactive proof editing mode, and an unpolished graphical frontend (`TkMic`) built almost entirely from the same building blocks. Neither user interface is particularly convenient for editing derivations, but the software suffices for simple examples. The programs are implemented in Objective CAML 1.06 and CamlTk 4.1. These are copyright 1997 INRIA, and are currently available free for personal use from `http://pauillac.inria.fr/ocaml/`.

The term and type unification algorithm is straightforward and based on one appearing in [18].The context unification algorithm is unsophisticated, but employs some simple strategies to identify obvious sources of unification failure early, such as different distributions of semicolons. The inference rules are also designed to limit the number of successful matches of context patterns where possible. For the derivations it has been used for, performance has been reasonable, especially considering that the executables are byte-code rather than native code.

Not all of the features of ICTT are currently implemented in the software. The treatment of arrays and variables is simplified in several respects. There are only variable and array *name* types, and no plain variable or array types. Since name types do not carry a canonical name as part of the type, as the variable and array types of Chapter 3 do, the get and put rules must refuse to work on variable or array names which are not raw parameters. For example, if a function returns a variable name as its result, the return value cannot be used as an argument to one of the variable rules. All variables and arrays must be referenced by their declared names. This is a severe restriction, but allows sufficient flexibility for deriving simple programs. Component-wise update is not supported in variables and arrays, and passive-discreteness checking is implemented as the incorrect shortcut that only examines the type of an object and rejects discrete function and universal types. As discussed in Chapter 3, this prevents certain kinds of important effect-compositions from being carried out, so the $\triangleright$To$\star$ rule is supplied in two varieties, one that insists on a passive-discrete type for A in $A \triangleright B$, and one that insists only on discrete type A. The second rule is documented as unsafe under conditions in which the A-component is an active term, that is, a variable name, an array name, or any term containing embedded assignment statements.

The checkers immediately verify that each newly-formed context is threadable and will reject any inference that forms an unthreadable context. These checks are performed by

applying the threading algorithm to a separate copy of the context, but the original is kept unthreaded in case the user has deliberately arranged the repeated identifiers in preparation for applying some inference rules.

4.3.1 A Trivial Example: the Identity Function on N

Before moving on to a more complicated example in Chapter 5, we revisit the fully-specified identity function first discussed in Chapter 1, and derive it for the natural numbers. The `mic` output, which echos the input commands, is shown in Figure 4.1.

The derivation requires only four steps:

1. the introduction of a parameter `aNat` which will be removed later through universal introduction;

2. the introduction of the (self-) equality predicate on `aNat` using equality introduction;

3. the abstraction of one side of the equality using existential introduction; and

4. the abstraction of the original parameter using universal introduction.

The resulting sequent for each step is shown by the checker in the comments following the inference commands. The final term

```
(FUN aNat: N. (+aNat, WEq(aNat)+))
```

is a function of universal type that returns an existential pair containing the function argument and the witness of its equality with itself. The last two inferences illustrate universal elimination as the mechanism for applying the function.The function application is not evaluated since the term language is not directly executable; it must be translated into an executable language first. Because the term calculus is largely functional despite the addition of variables and arrays, the simplest targets for translation are functional languages with imperative features like LISP or variants of ML. A translator to Objective CAML is under construction.

To produce the result of the traditional identity function without the specification carried by the existential, the derivation must be continued using existential-elimination to break up the existential pair, and then bang-elimination to discard the equality. Ideally, the system would be able to determine by itself which parts of a term are of computational interest and discard the rest; this is computational redundancy elimination and is not currently addressed in the MIZAR-I system.

```
module identity

(*  The identity function on naturals  *)
begin

nat by assumption ( nat: U ) .
 (* nat: U -> nat: U *)

(*  The equality  *)
[illegible] ( nat )
 (*  -> [illegible] nat . nat = nat *)

(*  [illegible]  *)
[illegible] by [illegible] ( nat, [illegible], [illegible] _x , [illegible] _x = nat ) .
 (* nat: U -> ( nat, [illegible] nat -> . [illegible] _x U . _x = nat *)

(*  The [illegible]  *)
identity by [illegible] ( [illegible] nat, [illegible] ) .
 (*
    ->
     [illegible] nat: U ( nat, [illegible] nat -> .
     [illegible] nat: U  [illegible] _x  U . _x = nat
   *)

(*  [illegible]  *)
[illegible] by assumption ( [illegible]: U ) .
 (* [illegible]: U -> [illegible]: U *)

(*  *)
[illegible] by [illegible] ( [illegible], identity ) .
 (*
   [illegible]: U ->
     [illegible] nat: U ( nat, [illegible] nat -> . [illegible]  [illegible] _x U . _x = [illegible]
   *)
```

Figure [illegible]: The identity function on the naturals

As we will see in the next chapter, the derivation of even very simple programs in [illegible] can require considerable effort. The system suffers from a lack of automation, and it takes practice and experience to use it well. As an illustration of the applicability of CTT in principle it suffices, but it is far from a production tool. Considerable improvements in user interface design, the management of intermediate proofs and proof information would be prerequisites for a practical system.

Chapter 5

An Example: Bubble-sort

In this chapter we attempt to use MIZAR-I to derive part of a Bubble-sort function on arrays of natural numbers, with an eye to evaluating MIZAR-I's potential as a practical tool. The derivation focuses on a function that exchanges two elements in an array, and illustrates how update operations are handled and how the contents of mutable locations are kept track of explicitly in the types. The derivation of this exchange function is crucial to developing the entire sorting function. We discuss how the rest of the derivation would proceed briefly at a high level.

Unfortunately, we conclude that MIZAR-I has a number of practical problems that limit its usefulness. The logic is clumsy and difficult to use, and the awkward interaction of the ! modality and $\triangleright$ connective makes it difficult to get assertions about the contents of arrays and variables in a convenient form. The derivation of even this small fragment is very long. Worst of all, the inference rules turn out to be unsound: it is possible to derive a type that makes a false statement about the contents of an array location. It may be possible to revise the rules so that the type theory is sound, but given the practical difficulties of deriving programs in MIZAR-I, the effort is probably of theoretical interest only.

We begin by recalling the bubblesort algorithm and then describe the helper function that the example will focus on, a routine that exchanges two elements in an array. This function involves a simple sequence of assignments that juggles the contents of array locations and a temporary variable, and so provides a small exercise of the features of MIZAR-I (and consequently the type theory ICTT) for deriving imperative code. We describe the major steps of the function and the corresponding MIZAR-I derivations, and the problems encountered on the way, and then finish with some comments on these problems.

5.1 The Bubble-sort Function

The version of Bubble-sort that we use here is based on one in [3]. The algorithm as stated there is

```
proc bubblesort1 (v: array, n: int) =
  numpairs := n;
  didswitch := true;
  while (didswitch) do
    numpairs := numpairs - 1;
    didswitch := false;
    for j := 0 to numpairs - 1 do
      if v[j+1] < v[j] then
        exch (v[j],v[j+1]);
        didswitch := true;
      end {if}
    end {for}
  end {while}
end {proc}
```

In this form, the program is not provable in MIZAR-I, because the only recursive combinator it supports is the primitive recursive (`for`-loop) combinator, so the `while`-loop cannot be derived. If we forsake some efficiency and dispense with the `didswitch` variable, we can rewrite the algorithm as

```
proc bubblesort2 (v: array, n: int) =
  for i := 0 to n-1 do
    for j := 0 to n - 1 - i do
      if v[j+1] < v[j] then
        exch (v[j],v[j+1]);
      end {if}
    end {for}
  end {for}
end {proc}
```

This version always performs $n(n+1)/2$ comparisons on any array, whereas this is true of the original algorithm only in the worst case[1].

5.2 The exch Function in Detail

The blow-up from lines of code to lines of proof can be large in MIZAR-I, sometimes a factor of 100. Details that are obvious to the programmer have to be spelled out in painful rigor to the proof-checker, and individual inferences usually advance the derivation only a small amount. For this reason, presenting an entire manual proof of Bubble-sort is infeasible, so we concentrate on the critical `exch` function, which takes an array `a` and two indices `i` and `j` and exchanges `a[i]` and `a[j]`. The pseudocode equivalent using MIZAR-I types and syntax is

```
fun exch (a: Arrayname[N,n],
          i: !N, g: !(i < n),
          j: !N, h: !(j < n)) .
(
   local temp: Varname[N];
   temp                 := @[a,Get(i),Get(g)];
   [a,Get(i),Get(g)] := @[a,Get(j),Get(h)];
   [a,Get(j),Get(h)] := @(temp);
)
```

where `i` and `j` are the indices and `g` and `h` are the corresponding evidence that `i` and `j` are in range. Both are supplied with banged types because they must be used more than once. We will usually write the more conventional looking `a[i]` instead of `[a,Get(i),Get(g)]`, but the more complicated form is what MIZAR-I expects. While tiny, the program uses both a variable and an array and alternates reads and assignments, so it is an ideal example for showing how effect-composition works.

5.2.1 Overview of the Proof

The three pseudocode assignment statements are viewed as six operations by MIZAR-I:

1. read the value of `a[i]`;

[1]Assuming the distribution of unsorted pairs is uniform, the first algorithm will do half as many comparisons as the second in the average case, so the modification doesn't change the average case asymptotic complexity class. But the modified algorithm will be slower in practice.

2. write this value into `temp`;

3. read the value of `a[j]`;

4. write this value into `a[i]`;

5. read the value of `temp`; and

6. write this value into `a[j]`.

The proof is best performed by grouping reads with their corresponding writes (as in the program itself) and performing three subderivations with the following general structure:

1. read the value of interest with an appropriate *get* rule; this delivers an existentially-quantified object together with an effect-statement that relates the object to the variable or array position from which it was read, and asserts the the operation did not change the value of the variable or array;

2. introduce a set of dummy assumptions, one for the object part, and one or more for the components of the effect part of the existential from step 1; these dummies will eventually be eliminated using the existential elimination rule with the original;

3. assign the dummy object to the target variable or array with an appropriate *put* rule; this produces a new effect statement;

4. introduce zero or more dummy assumptions corresponding to the parts of the effect-statement from step 3; these will be eliminated using multiplicative conjunction elimination with the original;

5. use the dummies from parts 2 and 4 to derive a new combined effect-statement; this process must be done with some care to ensure that a threadable arrangement of semicolons is achieved in the left context; and

6. combine this new effect-statement and the results of the original read and assignment in steps 1 and 3 using the appropriate elimination rules. This dispenses with the dummies from steps 2 and 4 and yields the result of the subderivation, the combined effect-statement for the read and write.

Applying the *get* and *put* rules is a one-inference exercise. The work is all in combining the effects. Once the three subderivations are completed, they are combined to produce the final effect-statement for the entire operation. This simultaneously constructs the term for

the exchange operation, which contains the three reads and assignments together with a considerable mass of evidence terms and structural baggage in the form of `let` and `letemp` definitions.

5.2.2 The Subderivations

We now discuss the particular subderivations in some detail. Each of them follows the basic structure outlined above. We will not go through the proof line-by-line, but highlight important or tricky parts. The complete text of the proof is contained in Appendix D.

The goal for the derivation is to obtain a sequent of the form

```
a, i, g, j, h    |-


   !(@[a,Get(i),Get(g)] = @[a',Get(j),Get(h)]) *
   !(@[a,Get(j),Get(h)] = @[a',Get(i),Get(g)]) *
   !(for i1: N. for g1: i1 < n.
          ( i1 < i | i < i1 ) * ( i1 < j | j < i1 ) -o
            @[a,i1,g1] = @[a',i1,g1] )
```

where the first conjunct states that the new (*un*primed) value of `a[j]` is equal to the old (primed) value of `a[i]`, that the new value of `a[i]` is equal to the old value of `a[j]`, and that for all indices other than `i` and `j` there has been no change. We adopt the convention in the MIZAR-I inference rules that old values appear on the right in equalities, and new ones on the left, to make remembering the meaning of a prime easier. Recall that the priming convention is relative, not absolute: primed objects represent state prior to the operation, and unprimed objects represent state after the operation. If an operation is the composition of two other operations, the combined effect-statement is usually obtained using the `>>-Intro (befi)` rule and then the `>>-To-* (bef2mc)` rule. The latter transforms the `>>` into a `*`, forces an order of evaluation on the components using `letemp` statements, and for each variable and array name in the first (earlier) effect-statement, adds an additional prime if it also appears in the second effect-statement. Equality elimination is then used to remove any identifiers with an intermediate number of primes, and then the final result has its primes compacted so there are only unprimed and singly-primed objects. For example, suppose we have the effect-statements

```
@(temp) = @[a',Get(j),Get(h)]
```

and

```
@[a,Get(i),Get(g)] = @(temp')
```

representing two operations, the first understood to have occurred before the second. Then joining these with a `>>` and employing `>>-To-*` we obtain

```
@(temp') = @[a'',Get(j),Get(h)] *
@[a,Get(i),Get(g)] = @(temp')
```

where both `temp` and `a` in the first component have received a new prime, since they both appear in the second component. This can then be reduced, using fresh dummy assumptions for the conjuncts and an equality elimination, to

```
@[a,Get(i),Get(g)] = @[a'',Get(j),Get(h)]
```

Then applying prime compaction changes the `a''` to an `a'`. That this represents the composition of two separate operations is hidden: the assertion is simply that `a[i]` is now whatever `a[j]` was beforehand.

For effect-statements that are banged or are universally-quantified propositions, they must first be unbanged and the quantifiers eliminated on dummy parameters. After combining the simplified effects as above, the quantifiers and bangs are then restored. This tearing-down and building-up accounts for a large proportion of the work in the proof.

Steps 1 and 2

The goal sequent for this subderivation is

```
a, i, g, temp   |-


  !(@(temp) = @[a',Get(i),Get(g)]) *
  !(for k: N. for f: k < n.
        @[a,k,f] = @[a',k,f]),
```

expressing that the value of the temporary variable after the operation is the same as the value at `a[i]` before, and that none of the array components changes value. Since `a` and `temp` do not interfere, we need not place a semicolon between `temp` and the rest. This part of the derivation is quite straightforward and follows the general pattern outlined above. The result of Step 1 is

```
a, i, g |-
  STEP1:
```

```
        ex __x176: N .
            !(__x176 = @[a', Get(i), Get(g)]) *
            !(for __x177: N . for __x178: __x177 < n .
                      (@[a, __x177, __x178] = @[a', __x177, __x178]))
```

where the double-underscore identifiers are dummies introduced by the checker. The quantified value, `__x176`, represents the value read, and the body says that this value is the same as `a[i]` before the read, and that for every legal index `k`, `a[k]` after the read is equal to `a[k]` before. The Step 2 portion of the derivation assumes dummies to stand in for `__x176` (`dSTEP1_ob`), the body of the existential (`dSTEP1_spec`), and separate subassumptions (`dSTEP1_spec_1` and `dSTEP1_spec_2`) for the body's two components. The dummy value `dSTEP1_ob` is written into the temporary variable using the regular *put* rule to produce the simple sequent

```
    dSTEP1_ob, temp |-
      dSTEP1_ob_in_temp: !(@(temp) = dSTEP1_ob).
```

The rest of the derivation is simple, though it takes several steps. The only minor technical wrinkle in it is that since the left context of `dSTEP1_ob_in_temp` is not banged, we must introduce another dummy assumption `ddSTEP1_ob_in_temp` for it so that the final equality we obtain can itself be banged. The equalities `dSTEP1_spec_1` and `ddSTEP1_ob_in_temp` are unbanged and combined using equality elimination to remove the dummy value `dSTEP1_ob`, resulting in

```
    dSTEP1_spec_1, ddSTEP1_ob_in_temp |-
       ai_eq_temp: @(temp) = @[a', Get(i), Get(g)]
```

This result is then banged and combined with the untouched universal `dSTEP1_spec_2` by $*$-introduction. Then the dummies `dSTEP1_spec_1` and `dSTEP1_spec_2` are replaced with `dSTEP1_spec` by $*$-elimination, and then at last `dSTEP1_ob` and `dSTEP1_spec` are replaced with the original result of Step 1 using existential elimination.

Steps 3 and 4

Ideally, the combined result of Steps 3 and 4 would be the sequent

```
    a, i, g, j, h   |-


      !(@[a,i,g] = @[a',j,h]) *
```

```
    !(for k:N. for f: k < n.
          ((k < i | i < k) -o @[a,k,f] = @[a',k,f])),
```

expressing that a[i] after is the same as a[j] before, and that all components other than the ith are unaffected. It turns out that we can derive a result which expresses the same properties, but with a slightly different structure:

```
a, j, h, i, g |-
  STEP4:
  !for i2: !(N) .
     for g2: !(Get(i2) < n) .
       @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
       Get(i2) < Get(i) | Get(i) < Get(i2) -o
          @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)].
```

The equality @[a,i,g] = @[a'',j,h] appears within the quantified statement instead of beside it, and the bang is on the outside, rather than on the individual components of the conjunction. As we shall see, the desired form cannot be obtained because of the restrictions imposed by the $\triangleright$To$\star$ rule.

The derivation begins as in the general pattern. Step 3 produces

```
a, j, h |-
  STEP3:
  ex __x205: N .
    !(__x205 = @[a', Get(j), Get(h)]) *
    !(for __x206: N .
        for __x207: __x206 < n .
          (@[a, __x206, __x207] = @[a', __x206, __x207]))
```

which is exactly the same result as for Step 1, except that a[j] is being read instead of a[i]. Again, a dummy is introduced for the quantified object (dSTEP3_ob) and for the components of the body (dSTEP3_eff, *etc.*), and the dummy object is written into position a[i]. Because the value read in fact depends on the array, and is being written back into the array, we must use the sequential *put* rule **asput** for the write, which produces

```
dSTEP3_ob; a, i, g |-
  assign_eff:
    !(@[a, Get(i), Get(g)] = dSTEP3_ob) *
```

```
        !(for __x210: N .
            for __x211: __x210 < n .
              (__x210 < Get(i)) | (Get(i) < __x210) -o
                @[a, __x210, __x211] = @[a', __x210, __x211]).
```

This sequent is eventually used to eliminate the dummy assumptions for the components of `assign_eff`, in a sequent whose context is structured so that `dSTEP3_ob` will appear together with the other `STEP3` dummy assumptions, and can then be replaced with the left context of `STEP3` above via an existential elimination. If the `assign_eff` sequent did not have the semicolon, the resulting context for the derivation would be

```
    a, j, h, a, i, g
```

which contains the array name `a` twice in the same independent context, and thus has a threading violation. The proof proceeds as for Steps 1 and 2, except that there are now several dummy assumptions corresponding to the components of the effect part of `STEP3` and `assign_eff`, and a reusable dummy index `i2:!N` and guard `g2:!(Get(i2) < n)` for removing the quantifiers. The strategy is to strip away the bangs on the conjuncts from `STEP3` and `assign_eff`, eliminate the quantifiers using the same index and guard, combine the bare equalities to eliminate `dSTEP3_ob` and the intermediate array `a'`, and then restore quantifiers and bangs where possible. Because the effects of Step 2 and Step 3 make statements about the same array they are sequenced using $\rhd$-introduction

```
    dSTEP3_eff_1, g2, i2, dSTEP3_eff_2; g2, i2, assign_eff_2,
      assign_eff_1
    |-
    new_eff_1:
      dSTEP3_ob = @[a', Get(j), Get(h)] *
      @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)]
      >>
      (Get(i2) < Get(i) | Get(i) < Get(i2)) -o
         @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)] *
         @[a, Get(i), Get(g)] = dSTEP3_ob
```

and then the result is converted to a multiplicative conjunction using the $\rhd$To$\star$ rule, and the repeated index and guard are eliminated by threading. This yields

```
    dSTEP3_eff_1, g2, i2, dSTEP3_eff_2, assign_eff_2, assign_eff_1
```

```
|-
new_eff_2:
  dSTEP3_ob = @[a'', Get(j), Get(h)] *
  @[a', Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
  *
  (Get(i2) < Get(i) | Get(i) < Get(i2)) -o
     @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)] *
     @[a, Get(i), Get(g)] = dSTEP3_ob
```

where the array a has picked up an extra prime in the first two conjuncts since it also appears in the last two conjuncts. Now, ▷To⋆ converts $A \triangleright B$ to $A' \star B$ provided A is discrete[2]. Unfortunately, banged and universal types are not discrete and this is why the bangs and quantifiers must be removed prior to converting to conjunctive form. The new conjunctive form is then simplified to eliminate dSTEP3_ob and a', producing

```
dSTEP3_eff_1, g2, i2, dSTEP3_eff_2, assign_eff_2, assign_eff_1
|-
new_eff_3:
  @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)]
  *
  Get(i2) < Get(i) | Get(i) < Get(i2) -o
    @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
```

There is no way to restore bangs to the individual conjuncts but rather only to the whole conjunction, and likewise the quantifiers for i2 and g2 will enclose the whole conjunction. This is a consequence of the discreteness restriction in ▷To⋆. The remainder of the proof comprises a series of eliminations to remove dummy assumptions.

Before leaving this section, we point out a subtle but serious problem with the sequential versions of the *put* rules for variables and arrays. Recall that above we said that since the value read depended on the array and was being written into the array, the vsput rule had to be used to introduce a semicolon. Because the vsput rule takes only a raw value, and not an existentially-quantified object with a value and the property it satisfies, there was actually nothing forcing us to combine the dSTEP3_eff and assign_eff dummies using a >> connective. We could have simply used a multiplicative conjunction, and derived an

[2]The MIZAR-I rule used is the dangerous form of ▷To⋆ because of the presence of a linear implication. As mentioned in Chapter 4, proper passivity-checking is not implemented in the checker.

incorrect result, because the necessary semicolon to separate the occurrences of `a` in the final sequent was already in place in the context for `assign_eff` and would then fall into position when `assign_eff_1` and `assign_eff_2` were eliminated. The array `a` would not have received an extra prime where necessary. Thus the rules as they stand permit us to derive incorrect specifications: the rules are unsound.

This might be fixed by changing the `put` rules so that they take only quantified values as inputs. The new rules with the resulting terms suppressed and with slight syntax changes over those presented in Chapter 3 have the forms shown in Table 5.1. Using these rules, all objects written into variables or arrays must be paired with their specifications or effects, and the sequencing of effects is forced from the outset. Combined effects can then be simplified using similar dummy-assumption techniques to those outlined here. Of course, these remarks are not a proof of soundness, but merely suggest that sound rules could be found; obviously further work with ICTT and MIZAR-I would require a rigorous soundness proof.

Steps 5 and 6

The goal for this subderivation is to produce the sequent

```
temp; a, j, h   |-


    !(@[a,Get(j),Get(h)] = @(temp)) *
    !(for j1: N. for h1: j1 < n.
            ((j1 < Get(j) | Get(j) < j1) -o @[a,j1,h1] = @[a',j1,h1]))
```

The techniques are essentially the same as in the first subderivation, except that it is a variable being read and an array being written. This time, we must be careful to put a semicolon between the variable `temp` and the array `a`, since we will need a semicolon to thread the occurrence of `a` in this sequent with the occurrences in the previous two results, and a semicolon to thread the two occurrences of `temp`. We are able to produce the sequent in the desired form.

Assembling the Final Effect

The results of the three subderivations are three effect-statements in isolation. We derive the final effect by tearing down effects to discrete form, sequencing, applying $\triangleright$To$\star$ to convert to conjunctive form and add primes where required, simplifying, restoring bangs and

$$\frac{\Gamma \vdash \exists x\!:\!A.\, P[x] \qquad \Delta \vdash v\!:\!\mathbf{varname}[A]}{\Gamma, \Delta \vdash \exists x\!:\!A.\, P[x] \star\, !(@v = x)}$$
(New Put)

$$\frac{\Gamma \vdash \exists x\!:\!A.\, P[x] \qquad \Delta \vdash v\!:\!\mathbf{varname}[A]}{\Gamma; \Delta \vdash \exists x\!:\!A.\, P[x] \triangleright\, !(@v = x)}$$
(New SPut)

$$\frac{\Delta \vdash \exists x\!:\!A.\, P[x] \qquad \Omega \vdash i : \mathrm{N} \qquad \Lambda \vdash g : i < n \qquad \Gamma \vdash a : \mathbf{arrayname}[A, n]}{\Delta, \Omega, \Lambda, \Gamma \vdash \exists x\!:\!A.\, P[x] \star\, !(@[a, i, g] = x) \star \ldots}$$
(New Array Put)

$$\frac{\Delta \vdash \exists x\!:\!A.\, P[x] \qquad \Omega \vdash i : \mathrm{N} \qquad \Lambda \vdash g : i < n \qquad \Gamma \vdash a : \mathbf{arrayname}[A, n]}{\Delta; \Omega, \Lambda, \Gamma \vdash \exists x\!:\!A.\, P[x] \triangleright\, !(@[a, i, g] = x) \star \ldots}$$
(New Array SPut)

Table 5.1: Better *put* rules.

quantifiers where possible, and finally eliminating the layers of assumptions to obtain the desired result. This part of the derivation is as long as the other three put together. Almost all of the work is in manipulating and simplifying formulas and obtaining the required sequent structures.

The final type for the exchange function is

```
exch:
  (for a: Arrayname[N, n] .
     for i: !(N) .
       for g: !(Get(i) < n) .
         for j: !(N) .
           for h: !(Get(j) < n) .
             for k: !(N) .
               for f: !(Get(k) < n) .
                 !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
                     ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                        @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
                    *
                    (((Get(k) < Get(i) | Get(i) < Get(k)) *
                        (Get(k) < Get(j) | Get(j) < Get(k)))
```

```
                    -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
```

Again, the final form for the body is not quite what we had hoped for. The case that i and j are unequal is left separate because performing a case analysis for the $i = j$ and $i \neq j$ cases requires copying the body and using up the copies of only some of the conjuncts. The result would be redundant, with two copies of some formulas. Because the individual conjuncts are not separately banged, they cannot be copied or disposed of individually. This awkward situation is a consequence of embedding the truths about terms into the language, so truth becomes a resource subject to the same book-keeping checks as anything else.

The final term is a couple of pages long and difficult read, but once redundant `let` definitions are eliminated by substitution, the assignment statements, which are protected in `letemp` definitions, appear in the correct sequence.

5.3 The Rest of the Proof

The remainder of the Bubble-sort proof requires a pair of induction (natural number elimination) applications to derive the inner and outer loops. The predicate $C[m : \mathbf{N}]$ (see the statement of the rule in Chapter 2) is the loop invariant, a property that is satisfied at the moment that the loop index is assigned value m. To derive a loop, we must decide on a candidate loop invariant C we believe it satisfies, prove $C[0]$, and then prove that $C[m] \multimap C[m+1]$ for arbitrary $m \in \mathbf{N}$. That is, that the invariant is satisfied upon entry to the loop, and if it is satisfied at the beginning of iteration m, then it will be satisfied at the beginning of the next iteration. The derivations of $C[0]$ and $C[m] \multimap C[m+1]$, together with the final value n of the loop index, form the arguments to the induction rule, and the result is $C[n]$. Naturally, C should be chosen so that $C[n]$ captures what the loop is supposed to accomplish. For loops that process arrays, the invariant is usually part effect-statement and part auxiliary specification.

The inner loop scans the range $\{0, \ldots, n-1-i\}$ and moves the maximum element in the range $\{0, \ldots, n-i\}$ into position $n-i$. At the start of iteration j we know that if j is in range then it is no less than any preceding element (vacuously true for $j = 0$)

```
for h: j < n.
    for i:N. for g: i < j.
        ex g1: i < n.
            @[a,i,g1] <= @[a,j,h]
```

and that the current state of the array is a permutation of its previous state:

```
ex p: N -o N.
  (for j: N. for k: N.
       (j < n) * (k < n) * ((p j) = (p k)) -o (j = k)) *
  (for j: N.
       (j < n) -o (ex k: N.
                        (k < n) * ((p k) = j))) *
  (for i: N. for g: i < n.
      (ex pg: (p i) < n.
          @[a, i, g] = @[a', (p i), pg]))
```

That is, there is a function on the naturals whose restriction to $0, \ldots, n-1$ is one-to-one and onto $0, \ldots, n-1$, and which maps each position in the new arrangement of the array to a position in the old arrangement holding the same value. The strict conjunction of these two propositions[3] forms the invariant proposition for the induction used to prove the inner loop. For the case $j = 0$ there is, strictly speaking, no previous arrangement of the array, but we can add a null-operation rule that allows us to obtain

```
!(for i: N. for g: i < n.
      @[a',i,g] = @[a,i,g])
```

any time it is required. Deriving the second part of the invariant is an imposing amount of work. Since the exchange function is certainly a permutation, and is the only operation used on the array in the inner loop, we might be tempted to introduce the invariant axiomatically with the **ax** rule. But this is less than satisfying, since we might get the axiom wrong and proceed to derive an incorrect program. While a practical formal method should allow the user to declare that something needs no proof, if only temporarily, it should also be simple enough to use that the proofs can be developed in reasonable time and with reasonable effort. This cannot be said for MIZAR-I.

Iteration i of the outer loop applies the inner loop to the index range $j = 0, \ldots, n-1-i$. At the beginning of the ith iteration of the loop, the subrange $\{n-i, \ldots, n-1\}$ is sorted (the range is empty for $i = 0$ and so the result holds vacuously), that is

```
for j: N. for gj: j < n.
for k: N. for gk: k < n.
   (n-i <= j) * (n-i <= k) * (j <= k) -o
```

[3]These types are not quite correct; for example, the function p is used repeatedly and so requires a bang. These details are left out because the regular first-order logic version of the proposition is easier to read.

```
                @[a, j, gj] <= @[a, k, gk]
```

and every element in the subrange is at least as great as every element in the lower subrange $\{0, \ldots, n-1-i\}$

```
        for j: N. for gj: j < n.
        for k: N. for gk: k < n.
            (n-i <= j) * (k < n-i) -o
                @[a,k,gk] <= @[a,j,gj]
```

and the arrangement of the array is a permutation of the previous arrangement, as above. The loop invariant for the inner loop on its final index value $j = n - 1 - i$ can be used to show the invariant holds for $i + 1$ if it held for i: if the last i elements were sorted, and at least as great as the first $n - i$ elements, then moving the largest element in the lower subrange to the last position in that subrange extends the upper sorted range by one position. The reader can check that the instantiation of the invariant for $i = n - 1$ asserts that the last $n - 1$ elements of the array are sorted and are all at least as great as the first element. From this we can conclude that the whole array is sorted.

5.4 Summary

Given the length of the derivation for the exchange function, the derivation of the full sorting algorithm is understandably large and difficult to manage. The inference rules have also been shown unsound, and a prerequisite for further work would be to revise the rules and develop a proof of soundness. Whether such work should be pursued, at least in the hope of developing a practical system, is debatable. Deriving programs in MIZAR-I is extremely expensive in terms of time and effort, and the book-keeping imposed by the logic prevents reasoning about terms in a natural way. The structure of left contexts must be planned in advance to avoid reaching dead-ends, and the logic is more complicated and brittle to use than traditional CTT. Perhaps a theorem prover for a repaired ICTT—with an intuitive user interface and enough power to discover routine or tedious derivations by itself—could be designed and built, but the structural idiosyncrasies of the logic make this extremely unlikely.

The concluding chapter summarizes the success of this project with respect to the goals stated in Chapter 1, and discusses some possible extensions to ICTT and the MIZAR-I software. It also suggests an alternative direction the work could take that may be more fruitful than the present one.

Chapter 6

Future Work and Conclusions

In the preceding chapters we developed ICTT, a constructive type theory for a λ-calculus with assignment, variables and arrays, and showed how this system could be used to derive fully-specified imperative programs, but with considerable difficulty. In this concluding chapter we assess our success by answering the three questions posed in the goals section of Chapter 1. We also summarize some additional constructs that could be added to ICTT to make it more full-featured, and suggest a different and perhaps more practical direction that this work could take.

6.1 Assessment of ICTT

We are now in a position to give answers to the questions posed in Chapter 1.

1. *Is it possible to develop a constructive type theory for a simple imperative programming language?*

   Almost certainly yes, although it may be so difficult that the exercise is questionable. We believe the problems with soundness discovered in Chapter 5 can be corrected, but have not provided a formal proof of this and so cannot claim certainty.

2. *Assuming it is possible, develop a simple prototype CTT for imperative programming. Is the resulting system easy to derive programs in?*

   No. ICTT is awkward to use. Using linear logic for this purpose makes writing proofs harder. ICTT is very brittle and it can be exceedingly difficult to get target types exactly. Succeeding with a derivation seems to take a great deal of advance planning. If anything, the programming becomes much less intuitive than in regular CTT, where the excruciating detail of derivations is already a hindrance.

3. *Could a practical tool for formal programming be built based on this CTT?*

   Almost certainly not. Automating CTT derivations is already difficult because automated theorem proving is hard to start off with, and the restriction to constructive inference rules makes finding proofs harder. The book-keeping restrictions on sequents in ICTT are probably an insurmountable obstacle. Even if this is not the case, our intuition is that an automatic theorem prover for the ICTT logic would be slow and probably useless for all but the simplest programs.

ICTT cannot be described as an operational success. A trivial three-line program took days to *almost* derive, and substantially larger programs are probably nearly impossible to tackle. Of course, the problem may be that ICTT is simply a very bad type theory for imperative programming, and there is a much simpler and more clever way to approach the problem. But we are unaware of other successful attempts.

Perhaps the best hope for CTT as a formal method is to stick to pure functional programming and then employ transformations to derived programs that make them imperative for the sake of efficiency.

This exercise has at least been instructive, and thinking about ICTT or similar systems may still have theoretical merits, or suggest approaches to building imperative programs that make them less tricky to reason about. In the remaining sections, we present some potential directions for future work.

6.2 Additional Imperative Constructions

The $\dagger$ modality and $\triangleright$ connective permit a richer set of imperative programming structures than just simple variables and arrays. In the following sections we summarize several additional constructions that would increase ICTT's flexibility, allow the development of abstract data types and modules, and provide a limited form of object-oriented programming.

6.2.1 Aliasable State

One of the important drawbacks of ICTT is that aliasing is not permitted for mutable objects, so it is impossible to build programs in which mutable data structures share members. This is a direct result of the interference control strategy which is necessary in order to ensure correct reasoning about the contents of variables and arrays. We can retain the ability to reason correctly about mutable objects and allow a restricted type of aliasing by

providing a different kind of effect information for objects that may be aliased. We show how through a simple example.

Suppose that we develop a program having a natural number variable x, and the value of x is always in the set $\{1, 2, 3, 4\}$. Suppose further that to prove the program correct all that need be known about x is that its value lies in this set. We never need to know, for example, after setting it to 3 that $x = 3$, but just that $1 \leq x$ and $x \leq 4$. That is, x satisfies the invariant $I[y] := 1 \leq y \star y \leq 4$. Now suppose that we could replace the accessor type of x,

$$\begin{aligned}\mathbf{N}\ \mathbf{var}\ [x] := \dagger(\quad & (\exists v : \mathbf{N}.\, !(*x = v) \star !(*x = *x')) \ \sqcap \\ & (\forall v : \mathbf{N}.\, !(*x = v)) \quad),\end{aligned}$$

with the type

$$\begin{aligned}&(\mathbf{N}, 1 \leq x \star x \leq 4)\,\mathbf{state}[x] := \\ &\qquad !(\quad (\exists v : \mathbf{N}.\, !(*x = v) \star !(1 \leq v \star v \leq 4)) \ \sqcap \\ &\qquad\quad (\forall v : \mathbf{N}.\, !(1 \leq v \star v \leq 4) \multimap !(1 \leq *x \star *x \leq 4)) \quad),\end{aligned}$$

where the types of the get and put components have changed. The type of the get component expresses that whatever the value of x, we have $I[*x]$. The type of the put component indicates that given any natural number v and a proof of $I[v]$, we can assign v to x and $I[*x]$ still holds. With effects restricted to the invariant property, we are no longer able to reason about what particular value x has at any time; but we are also free to alias x, because the invariant can never be violated by an update of x. This accounts for the ! modality in the new type instead of the † modality.

We can generalize the **state** type as we did variables and arrays. Let A be a passive discrete type and $I[x_{\nu_1} : A_{\nu_1}, \ldots, x_{\nu_n} : A_{\nu_n}]$ a predicate type over some subcollection of the component types Δ_A. We provide a new discrete type (A, I) **statename** and a corresponding accessor type (A, I) **state**[(A, I) **statename**] with the interpretation

$$\begin{aligned}!(\quad & (\exists v : A.\, !(*x = v) \star I[v_{\nu_1}, \ldots, v_{\nu_n}]) \ \sqcap \\ & (\forall v : A.\, I[v_{\nu_1}, \ldots, v_{\nu_n}] \multimap I[*_{\nu_1}x, \ldots, *_{\nu_n}x]) \quad)\end{aligned}$$

where v_{ν_i} is component ν_i of v. This is not the most flexible definition of $(A, I)\,\mathbf{state}[x]$, because x can only be updated by supplying all its components at once, and all components must be retrieved at once. A better approach would allow components that are not mentioned in the invariant to be read and written exactly as they are in A **var** $[x]$, and require the other components to be read and written as a group. Further refinement might allow reasoning about the value of a component of x within a thread between an update of x and the next update of any $v : (A, I)\,\mathbf{state}[x]$.

6.2.2 Abstract Data Types and Objects

ICTT has no integer type, but one could be implemented using pairs of naturals, one component p for the nonnegative part, the other n for the negative part. To make it easy to compare integers, the components should be kept in a normalized form characterized by the rules

- if $p < n$ then $(p, n) := (0, n - p)$
- if $n < p$ then $(p, n) := (p - n, 0)$
- if $n = p$ then $(p, n) := (0, 0)$

Obviously, a normalization operation $\mathbf{norm} : (\mathbf{N} \star \mathbf{N})\,\mathbf{var}\,[x] \multimap (*_1 x = 0 \sqcup *_2 x = 0)$ could be implemented using the get and put components of $(\mathbf{N} \star \mathbf{N})\,\mathbf{var}\,[x]$. Given **norm**, a suite of functions implementing common integer operations that perform destructive update on (for example) the first argument could be derived. Defining **Int** as

$$\exists x : \mathbf{N} \star \mathbf{N}.\,(x_1 = 0 \sqcup x_2 = 0),$$

Intvarname as **Int varname** and $\mathbf{Intvar}\,[x : \mathbf{Intvarname}]$ as $(\mathbf{Int})\,\mathbf{var}\,[x]$ the type of an addition function on $\mathbf{Intvar}\,[x]$ would be

$$\mathbf{add} : \mathbf{Intvar}\,[x] \multimap \mathbf{Intvar}\,[y] \multimap \mathbf{IsSum}\,[*x, *x', *y]$$

where $\mathbf{IsSum}\,[x, y, z]$ indicates that x is the integer sum of y and z and has the rather tedious definition[1]

$$\begin{array}{l}
((0 \leq y_1) \star (0 \leq z_1) \multimap (x_1 = y_1 + z_1) \star (x_2 = 0)) \star \\
((0 < y_2) \star (0 < z_2) \multimap (x_2 = y_2 + z_2) \star (x_1 = 0)) \star \\
((y_1 \leq z_2) \multimap (x_1 = 0) \star (x_2 = z_2 - y_1)) \star \\
((z_2 < y_1) \multimap (x_1 = y_1 - z_2) \star (x_2 = 0)) \star \\
((z_1 \leq y_2) \multimap (x_1 = 0) \star (x_2 = y_2 - z_1)) \star \\
((y_2 < z_1) \multimap (x_1 = z_1 - y_2) \star (x_2 = 0))
\end{array}$$

which covers the cases where y and z are both nonnegative, both negative, and the mixed cases with $|y| < |z|$ and *vice-versa*.

Now let $f : \mathbf{Intvar}\,[x] \multimap A \multimap B$ be a function that takes an $\mathbf{Intvar}\,[x]$, an argument of type A and produces a result of type B. Suppose that ICTT allowed us to expand $\mathbf{Intvar}\,[x]$ to its full definition; then we could apply f to an accessor $v : \mathbf{Intvar}\,[x]$ to obtain $(f\,v) : A \multimap B$, and then add the new function $(f\,v)$ to the type $\mathbf{Intvar}\,[x]$ by

[1]We have abused notation here somewhat by using, for example, x_1 where we should use $(\mathrm{Fst}\,(x))$, but the intention should be clear.

1. expanding **Intvar**$[x]$ to its underlying type $\dagger(\ldots)$;

2. applying $\dagger$E to obtain the additive conjunction

$$\{*_1, *_2, :=_1, :=_2\} : (\exists p : \mathbf{N}.\ldots \sqcap \forall n : \mathbf{N}.\ldots)$$

   of get and put operations;

3. forming a new conjunction

$$\{*_1, *_2, :=_1, :=_2, (f\,v)\} : (\exists p : \mathbf{N}.\ldots \sqcap \forall n : \mathbf{N}.\ldots \sqcap A \multimap B)$$

   consisting of the old operations together with the new one; and

4. restoring the $\dagger$ modality with $\dagger$I.

The new type is an enhanced **Intvar**$[x]$ with an additional operation of type $A \multimap B$. This procedure could be used to enhance **Intvar**$[x]$ with the **norm** operation and the rest of the suite of integer operations. Moreover, since we can project out arbitrary sub-conjunctions of an additive conjunction, we can also discard the original get and put components (assuming we have provided normalizing counterparts or other ways to update the value) so that what remains are just the integer operations. The new type represents an accessor for (a particular implementation of) an abstract integer type. We can imagine an extended ICTT in which there are facilities for manipulating types called **extend** and **restrict**, where **extend** forms a new imperative accessor type from an existing one and one or more additional operations, and **restrict** forms a new imperative accessor type from an existing one by discarding some of its operations. A rich system would allow the abstract declaration of a new applicative type **Int** and some abstract predicates and operations over it, and a corresponding imperative type whose effects are expressed in terms of the abstract predicates. There would also be facilities for implementing the type and its operations from other types. For example the syntax of a declaration in a computer implementation of the new system might be something like

```
type Int;                              // the new applicative type
const Zero: Int;                       // constants
const One:  Int;
type IntEq[x:Int, y:Int];              // abstract predicates
type IntSum[x:Int, y:Int, z:Int];
type IntDiff[x:Int, y:Int, z:Int];
```

```
...
// axioms that express the relationship between the abstract predicates
prop IntProp1: IntEq[x,y] * IntDiff[z,x,y] -o IntEq[z,Zero];
...

// applicative versions of operations
prop IntAdd: forall x: Int. forall y: Int. exists z: Int. IntSum[z,x,y];
prop IntSub: forall x: Int. forall y: Int. exists z: Int. IntDiff[z,x,y];
...

name type Intvarname;                   // ``pointer'' type for an
                                        // imperative counterpart

accessor type Intvar[x:Intname]         // an accessor
with components
        // component operations, with result types possibly expressed
        // in terms of abstract predicates on Ints
        get: exists y:Int. IntEq[*x,y],
        set: forall y:Int. IntEq[*x,y],
        varset: forall y:Intname. IntEq[*x,*y],
        varadd: forall y:Intname. IsSum[*x,*x',*y],
        varsub: forall y:Intname. IsDiff[*x,*x',*y],
        ...;
```

The suggested implementation would define `Int` and `Intname` as above, and `Intvar[x]` as

```
(Int) var [x] extended with
        // operations to add
        ...
and restricted to
        // operations to keep,
```

define the abstract predicates using predicates on $\mathbf{N} \star \mathbf{N}$ component types, and define the `Intvar[x]` operations using $(\mathbf{N} \star \mathbf{N})$ **var** $[x]$ component operations.

Type extension, or a modified form of it, could underlie a kind of object-oriented type system. If extension is the mechanism used to produce a new accessor type B from an

old type A, then B is a subtype of A and terms of type B could be used as terms of type A. There are a number of issues, such as method naming, overriding and virtual methods, which are not addressed and are necessary for a true OO system, but the possibilities here warrant further exploration.

6.2.3 Modules

One last additional imperative construction is a module type. A module is a conjunction

$$\dagger(I_1 \sqcap \ldots \sqcap I_n) \star !(A_1 \sqcap \ldots \sqcap A_m)$$

where each I_j is an imperative type (variable, array, function with side-effects, *etc.*) and each A_k is an applicative type. The imperative portion is made reusable through the $\dagger$, and the applicative part is made reusable through the !. As with the other constructions, modules should have their own sugared forms and high-level constructions for manipulating them. If ICTT is further extended with some higher-order features, transformations on modules akin to the functors of Standard ML could be implemented.

6.3 Additional Features

Even from a functional programming perspective, ICTT lacks a number of important features. A full-featured language should include a wider range of base types such as floating-point numbers, and characters and strings. The addition of well-founded types, making lists and trees available, is desirable. Certainly more general recursive combinators should be added to allow the programming of while-loops or equivalent constructions, and recursive functions, especially for traversing well-founded types. And, of course, input and output facilities would be welcome.

6.4 An Alternative: Type-Directed Program Verification

An alternative use of Reddy's extended linear logic abandons the idea of constructive development of programs, but exploits the fact that as a type system the logic offers control over interference and an explicit sequencing operator. As was said in Chapter 3, reasoning about imperative programs requires knowing what terms interfere and the sequence in which interfering computations are performed. If instead of deriving programs we simply program them and then reason about them after the fact, we are free to use whatever logic we wish, as long as there is a systematic technique for composing what we know about the

parts of a program into what we know about the whole program that takes interference and sequencing into account. Perhaps a linear logic type system for a language could help provide this systematic framework.

For example, consider a programming language in which, instead of just expressions and types, the fundamental construction in programs is a triple comprising an expression t, a type A, and a specification σ, written

$$t : A \bullet \sigma.$$

The specifications would be written in some convenient logic; suppose first-order logic for this example. When variables are declared, they are not only given a type, but a specification as well[2]. The typing rules would be formalized in the extended linear logic or some related logic, and the typing rules would not only say how new terms are typed using the types of subterms, but how the specifications of terms are constructed from those of subterms. An assignment statement for a variable might have the rule

$$\frac{\Gamma \vdash t : A \bullet \sigma_1 \qquad \Delta \vdash v : A\mathbf{var} \bullet \sigma_2}{\Gamma, \Delta \vdash (v := t) : \mathbf{1} \bullet (@v = t) \wedge \sigma_1 \wedge \sigma_2}$$

for the non-sequential version, where the new value does not depend on v, and the rule

$$\frac{\Gamma \vdash t : A \bullet \sigma_1 \qquad \Delta \vdash v : A\mathbf{var} \bullet \sigma_2}{\Gamma ; \Delta \vdash (v := t) : \mathbf{1} \bullet (@v = t) \wedge \sigma_1' \wedge \sigma_2}$$

for the sequential version, where σ_1' is the primed version of the specification σ_1, and the priming convention is analogous to that developed for ICTT. The rules say that the specification for the assignment takes whatever is true of the value (which may contain some truth about the variable) and conjoin it with the specification for the variable and the statement that the variable has a new value, possibly putting it in the relative past by priming. The type system ensures that the specification for the value is in fact primed when it needs to be.

We do not elaborate a full set of rules for this programming language. Programs are written an in any other language, but are annotated with specifications. The specifications are composed in a way that automatically primes identifiers when there is potentially more than one version of a value being reasoned about. Assuming some of the extensions proposed earlier in this chapter for handling aliased state, the rules could prevent overly-specific reasoning about aliased objects that could be incorrect. The exchange function example

[2]For most variables, the specification might be empty (TRUE), but for some it could express constraints like "v is always between 0 and 7").

of Chapter 5 would then be easy to write; the specification that is automatically built would have to be reduced to a simpler form, but the programming would be simple, and the specifications much more amenable to manipulation by an automatic system. This application of linear logic to imperative program verification is probably worth pursuing further.

6.5 Summary

The goal of this thesis was to show that there is a constructive type theory for imperative programming. The ICTT system provides a correspondence between an imperative λ-calculus and an enhanced linear logic, and allows programs to be developed in tandem with their specifications. Variable and array constructions that closely mimic those in typical imperative programming languages were developed, and the system could conceivably be extended with further constructions representing abstract data types, objects, and modules. Unfortunately, substantial automated support is probably impossible, and this fact alone ensures that ICTT will remain a theoretical curiosity. Perhaps a more fruitful application of the extended linear logic type system would be a kind of type-directed program verification that makes specification external to the language, but allows automatic construction of specifications from component specifications.

Bibliography

[1] Abelson and Sussman. *Structure and Interpretation of Computer Programs.* 2nd ed. Cambridge, Massachusetts: MIT Press, 1996.

[2] Abramsky, Samson. *Computational Interpretations of Linear Logic. Theoretical Computer Science*, **111**, 1-2 (1993), 3–57.

[3] Baase, Sara. *Computer Algorithms: Introduction to Design and Analysis.* 2nd ed. Reading, Massachusetts: Addison-Wesley, 1991.

[4] Baker, Henry G. *Lively Linear Lisp — "Look Ma, No Garbage!"*. ACM *Sigplan Notices* **27**, 8 (1992), 89–98.

[5] Benton, Nick and Philip Wadler. *Linear Logic, Monads and the Lambda Calculus.* In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1996, 420–431.

[6] Garrigue, Jacques. *Introducing Stateful Objects in a Transformation Calculus.* In *Proceedings of the JSSST Workshop on Object-Oriented Computing*, Hakone, 1993. URL: `http://web.yl.is.s.u-tokyo.ac.jp/members/garrigue/papers/wooc.html`.

[7] Garrigue, Jacques. *The Transformation Calculus and its Typing.* In *Proceedings of the Workshop on Type Theory and its Applications to Computer Systems*, Kyoto University Research Institute in Mathematical Sciences, July 1993. URL: `http://web.yl.is.s.u-tokyo.ac.jp/members/garrigue/papers/tcalc.html`.

[8] Garrigue, Jacques and Hassan Ait-Kaci. *The Typed Polymorphic Label-Selective Lambda-Calculus.* In *Conference record of the Twenty-second ACM Symposium on Principles of Programming Languages.* ACM Press, 1994, 35–47.

[9] Girard, Jean-Yves. *Linear Logic.* Theoretical Computer Science, 50(1987) 1-102.

[10] Goguen, Joseph. *Semantics of Imperative Programs.* Cambridge, Massachusetts: MIT Press, 1996.

[11] Hudak, Paul. *Mutable Abstract Datatypes, or, How to Have Your State and Munge it Too.* Yale University technical report YALEU/DCS/RR-914, 1993.

[12] Kogge, Peter M. *The Architecture of Symbolic Computers.* New York: McGraw-Hill, 1991.

[13] Knight, Brent. *Safe strict evaluation of redundancy-free programs from proofs.* M.Sc. Thesis, University of Victoria, 1994.

[14] Lafont, Yves. *The Linear Abstract Machine.* Theoretical Computer Science, 59(1988), 157-180.

[15] Lincoln, Patrick. *Linear Logic.* ACM *SIGACT Notices*, **23**, 2 (1992), 29–37.

[16] Mackie, Ian. *Lilac: a Functional Programming Language Based on Linear Logic.* Master's Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1991.

[17] Odersky, Martin and Dan Rabin, Paul Hudak. *Call-by-Name, Assignment, and the Lambda Calculus.* In *Conference record of the Twentieth Annual ACM Symposium on Principles of Programming Languages.* ACM Press, 1993, 43–56.

[18] Paulson, L.C. *ML for the Working Programmer.* 2nd ed. Cambridge: Cambridge University Press. 1996

[19] Peyton-Jones, Simon and Philip Wadler. *Imperative functional programming.* University of Glasgow research report FP-93-01, 1992.

[20] Reddy, Uday. *A Linear Logic Model of State (Preliminary Report).* University of Illinois at Urbana-Champaign technical report, July, 1992.

[21] Reddy, Uday. *A Linear Logic Model of State.* University of Glasgow research report FP-1993-3, 1993.

[22] Scedrov, Andre. *A Brief Guide to Linear Logic.* In G. Rozenberg and A. Salomaa (eds), *Current Trends in Theoretical Computer Science.* World Scientific, 1993, 377-394.

[23] Scedrov, Andre. *Linear Logic and Computation: a Survey.* In H. Schwichtenberg (ed), *Proof and Computation, Proceedings of the Marktoberdorf Summer School 1993.* Springer-Verlag, 1993, 281–298.

[24] Swarup, Vipin. *Type Theoretic Properties of Assignments.* University of Illinois at Urbana-Champaign technical report UIUCDCS-R-92-1777, 1992.

[25] Swarup, Vipin and Uday Reddy. *A Logical View of Assignments.* In J. Paul Meyers and Michael J. O'Donnell (eds), *Proceedings of Constructivity in Computer Science* (LNCS 613). Springer-Verlag, 1992, 131–149.

[26] Swarup, Vipin and Uday Reddy, Evan Ireland. *Assignments for Applicative Languages.* In John Hughes (ed), *Functional Programming Languages and Computer Architecture* (LNCS 523). Springer Verlag, 1991, 192–214.

[27] Thompson, Simon. *Type Theory and Functional Programming.* Wokingham: Addison Wesley, 1991.

[28] Troelstra, Anne S. *Lectures in Linear Logic.* Stanford: CSLI, 1992.

[29] Wadler, Philip. *A taste of linear logic.* In Andrzej M. Borzyszkowski and Stefan Sokolowski (eds), *Eighteenth International Symposium on Mathematical Foundations of Computer Science* (LNCS 711). Springer-Verlag, 1993, 185-210.

[30] Wadler, Philip. *Comprehending monads.* University of Glasgow technical report, 1992.

[31] Wadler, Philip. *How to declare an imperative.* ACMCS **29**, 3 (1997), 240–263.

[32] Wadler, Philip. *Is there a use for linear logic?.* In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 91).* ACM Press, 1991.

[33] Wadler, Philip. *Linear types can change the world!.* In M. Broy and C. Jones (eds), *IFIP TC 2 Working Conference on Programming Concepts and Methods.* North Holland, 1990, 347–359.

Appendix A

Imperative Program Transformations: Loops

Even in classical CTT if we have support for *primitive recursion* we can construct loops. Primitive recursion is typically introduced via an elimination rule for an inductive type like the natural numbers. For example Simon Thompson [27] introduces the naturals $\mathbf{N}$ to a classical CTT and provides the elimination rule:

$$\frac{n : \mathbf{N} \quad c : C[0/x] \quad f : (\forall n : \mathbf{N}).(C[n/x] \rightarrow C[(succ\, n)/x])}{\lambda n : \mathbf{N}.(prim\, n\, c\, f) : (\forall n : \mathbf{N}).C[n/x]}$$

where the function *prim* is the *primitive recursor*. The computation rules for *prim* are

$$\begin{array}{rcl} prim\, m\, c\, f & \Rightarrow & c \\ prim\,(succ\, n)\, c\, f & \Rightarrow & f\, n\,(prim\, n\, c\, f) \end{array}$$

Writing out a couple of example cases shows that

$$\begin{array}{rcl} prim\, 1\, c\, f & \Rightarrow & f\, 0\, c \\ prim\, n\, c\, f & \Rightarrow & f\,(n-1)\,(f\,(n-2)\,(\ldots(f\, 0\, c)\ldots)). \end{array}$$

Every function that can be written in terms of the primitive recursor *prim* can be written in terms of a primitive *tail* recursor *tprim* of type

$$tprim : \mathbf{N} \rightarrow C \rightarrow (\mathbf{N} \rightarrow C \rightarrow C) \rightarrow \mathbf{N} \rightarrow C \rightarrow C$$

given by

$$\begin{array}{rcll} tprim\, n\, c\, f\, 0\, v & := & v & \\ tprim\, n\, c\, f\,(m+1)\, v & := & tprim\, n\, c\, f\, m\,(f\,(n-m-1)\, v) & \text{if } m < n \\ & := & v \quad \text{otherwise} & \end{array}$$

The equivalence is given by

$$prim\, m\, c\, f = tprim\, n\, c\, f\, n\, c$$

for any choice of n, c and f. Bounded tail recursion can of course be systematically converted into bounded iteration. Observe that we can rewrite the rule for evaluation *tprim* expressions as follows:

$$\begin{array}{rcll} tprim\, n\, c\, f\, m\, v & := & tprim\, n\, c\, f\,(m-1)\,(f\,(n-m)\, v) & \text{if } 1 \leq m \leq n \\ & := & v \quad \text{otherwise} & \end{array}$$

Thus we can express the computation $tprim\, n\, c\, f\, n\, c$ using the loop

```
m := n;
v := c;
while (1 <= m <= n) do
        v := f (n-m) v;
        m := m - 1;
end;
```

Here the variables m and v are temporaries that are introduced solely for the purposes of the computation. The "return" value at the end of the loop is in v.

So for loop control structures can be introduced even in a purely functional context: the mutable variables are only temporaries that are not directly under the programmer's control. This kind of transformation is regularly performed as an optimization in functional language compilers.

Since for loops can be introduced using temporary variables even in a purely functional context, we can view the choice between for loops and bounded tail-recursion as a transformation or optimization issue. From the CTT point of view, the control flow of imperative languages is not difficult to achieve. What is difficult is dealing with terms that change their meaning as the program executes.

Appendix B

Full Linear Logic

This appendix presents the inference rules for full linear logic and introduces the coherence space semantics. It summarizes [28].

B.1 Inference Rules

Axiom and Cut rules

$$A \vdash A \quad (\mathrm{Ax}) \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \quad (\mathrm{Cut})$$

Propositional part

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \; (\neg\mathrm{L}) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \; (\neg\mathrm{R})$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \sqcap B \vdash \Delta} \; (\sqcap\mathrm{L}_1) \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \sqcap B, \Delta} \; (\sqcap\mathrm{R})$$

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \sqcap B \vdash \Delta} \; (\sqcap\mathrm{L}_2)$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \star B \vdash \Delta} \; (\star\mathrm{L}) \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \star B, \Delta, \Delta'} \; (\star\mathrm{R})$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \sqcup B \vdash \Delta} \; (\sqcup\mathrm{L}) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \sqcup B, \Delta} \; (\sqcup\mathrm{R}_1)$$

$$\frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \sqcup B, \Delta} \; (\sqcup\mathrm{R}_2)$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A + B \vdash \Delta, \Delta'}\ (+\mathrm{L}) \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A + B, \Delta}\ (+\mathrm{R})$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \multimap B \vdash \Delta, \Delta'}\ (\multimap\mathrm{L}) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \multimap B, \Delta}\ (\multimap\mathrm{R})$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, \mathbf{1} \vdash \Delta}\ (\mathbf{1}\mathrm{L}) \qquad \vdash \mathbf{1}\ \ (\mathbf{1}\mathrm{R})$$

$$(\text{No } \top\mathrm{L}) \qquad \Gamma \vdash \top, \Delta\ \ (\top\mathrm{R})$$

$$\mathbf{0} \vdash\ \ (\mathbf{0}\mathrm{L}) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \mathbf{0}}\ (\mathbf{0}\mathrm{R})$$

$$\Gamma, \bot \vdash \Delta\ \ (\bot\mathrm{L}) \qquad (\text{No } \bot\mathrm{R})$$

Quantifier rules (x not free in t, y not free in Γ, Δ)

$$\frac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x.\, A \vdash \Delta}\ (\forall\mathrm{L}) \qquad \frac{\Gamma \vdash A[y/x], \Delta}{\Gamma \vdash \forall x.\, A, \Delta}\ (\forall\mathrm{R})$$

$$\frac{\Gamma, A[y/x] \vdash \Delta}{\Gamma, \exists x.\, A \vdash \Delta}\ (\exists\mathrm{L}) \qquad \frac{\Gamma \vdash A[t/x], \Delta}{\Gamma \vdash \exists x.\, A, \Delta}\ (\exists\mathrm{R})$$

Rules for exponentials (modalities)

$$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (!\mathrm{L}) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (!\mathrm{L})$$

$$\frac{!\Gamma \vdash A, ?\Delta}{!\Gamma \vdash !A, ?\Delta}\ (!\mathrm{R}) \qquad \frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (!\mathrm{C})$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash ?A, \Delta}\ (?\mathrm{W}) \qquad \frac{!\Gamma, A \vdash ?\Delta}{!\Gamma, ?A \vdash ?\Delta}\ (?\mathrm{L})$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash ?A, \Delta}\ (?\mathrm{R}) \qquad \frac{\Gamma \vdash ?A, ?A, \Delta}{\Gamma \vdash ?A, \Delta}\ (?\mathrm{C})$$

B.2 Coherence Space Semantics

The original semantics for linear logic was given in terms of coherence spaces. More recent abstract semantics have been categorical. This section gives the necessary definitions to follow the presention in Chapters 2 and 3.

Definition 3 A *web* is a set $\mathcal{A}$, whose elements are called *atoms*, and a reflexive, symmetric relation $\sim$ called *coherence*.

A subset $A \subseteq \mathcal{A}$ is called coherent if every pair of atoms in A is coherent. In particular, the empty set and all singleton subsets of A are coherent. We write **Coh** $(\mathcal{A}, \sim)$ for the coherence space on $(\mathcal{A}, \sim)$. Obviously, any subset of a coherent set is itself coherent.

The *coherence space* associated with a web $(\mathcal{A}, \sim)$ is the collection of coherent subsets of $\mathcal{A}$ ordered by inclusion. If $\sim$ is equality, the coherence space is called *discrete.*

A discrete coherence space contains only the empty set and the singleton sets of atoms. Two special coherence spaces are $\underline{0}$, the coherence space on the web formed by the empty set and the empty relation. and $\underline{1}$, the coherence space on the web formed by a singleton set and equality. For the purposes of this thesis we will call $\underline{1}$ a *single-atom* coherence space. When given a coherence space A we will write $\mathcal{A}_A$ for the atoms of A.

Definition 4 Let A and B be coherence spaces. A function $f : A \to B$ is called *linear* iff whenever $\beta \in f(a)$, there is an $\alpha \in a$ such that $\beta \in f(\{\alpha\})$.

Lemma 4 *A map $f : A \to B$ is linear iff it has the following properties:*

1. *$a \cup a' \in A \Rightarrow f(a \cap a') = f(a) \cap f(a')$ (f commutes with intersections of subsets of coherent sets)*
2. *if $X \subset A$ and for all $b, c \in X$ we have $b \cup c \in A$, then $f(\bigcup X) = \bigcup\{f(b) : b \in X\}$ (f commutes with arbitrary unions of subsets of coherent sets)*

Definition 5 Associated with every linear function $f : A \to B$ is a subset of $\mathcal{A}_A \times \mathcal{A}_B$ called the *linear trace of f*, ltr_f with the properties

1. $(\alpha, \beta) \in \mathrm{ltr}_f \Rightarrow \beta \in f(\{\alpha\})$;
2. for every $(\alpha, \beta), (\alpha', \beta') \in \mathrm{ltr}_f$, $\alpha \sim_A \alpha' \Rightarrow \beta \sim_B \beta'$; and
3. if $(\alpha, \beta), (\alpha', \beta) \in \mathrm{ltr}_f$, then $\alpha \sim_A \alpha' \Rightarrow \alpha = \alpha'$.

Conversely, any set $X \subseteq \mathcal{A}_A \times \mathcal{A}_B$ that satifies these properties (with ltr_f replaced by X) determines a linear function from A to B.

In the coherence space semantics for LL, each proposition represents a coherence space; to each connective we assign a coherence space derived from the coherence spaces for the inputs. In particular

Definition 6 *The coherence space $A \multimap B$ is that for the web $(\mathcal{A}_A \times \mathcal{A}_B, \sim_{A \multimap B})$ where*

$$(\alpha, \beta) \sim_{A \multimap B} (\alpha', \beta') := \alpha \sim_A \alpha' \Rightarrow (\beta \sim_B \beta' \wedge (\beta = \beta' \Rightarrow \alpha = \alpha'))$$

The coherence relation $\sim_{A \multimap B}$ has an important property:

Lemma 5 *A subset of $\mathcal{A}_A \times \mathcal{A}_B$ is a linear trace iff it satisfies the coherence relation $\sim_{A \multimap B}$.*

Thus every linear function between A and B has a (unique) correponding element in $A \multimap B$. Moreover, any pairing of atoms of A and atoms of B that respects $\sim_{A \multimap B}$ defines a linear function from A to B, and this can be used to justify the inference rules of linear logic. This is exploited in the section on connectives in Chapter 2.

The remaining connectives can also be given corresponding webs whose coherence spaces are consistent with the inference rules of LL. The webs presented in [28] are shown in table B.2. **Fincoh** (A) is the set of finite elements of A (*i.e.* the finite coherent subsets of $\mathcal{A}_A$).

Chapter 3 uses the notions of discrete and single-atom types. These, as the names suggest, are types whose corresponding coherence spaces are discrete or single-atom. In general the connectives of linear logic do not preserve discreteness or single-atomness. The conditions under which these properties are preserved for the connectives used in LCTT are given in Table B.1.

Type	Discrete when	Single-atom when
$A \star B$	A and B discrete	A and B single-atom
$A \sqcap B$	never	never
$A \sqcup B$	A and B	never
$A \multimap B$	A single-atom, B discrete	A and B single-atom
$!A$	never	never
$\exists x : A.\, P[x]$	A and $P[x]$ discrete	A and $P[x]$ single-atom
$\forall x : A.\, P[x]$	A single-atom, $P[x]$ discrete	A and $P[x]$ single-atom

Table B.1: Discrete and single-atom types

Negation

$$\sim A := \mathbf{Coh}\left(\mathcal{A}_A, \sim_{\sim A}\right) \text{ where } x \sim_{\sim A} y := x \not\sim y \vee x = y$$

Multiplicative Conjunction

$$A \star B := \mathbf{Coh}\left(\mathcal{A}_A \times \mathcal{A}_B, \sim_{A\star B}\right) \text{ where } (a,b) \sim_{A\star B} (a',b') := a \sim_A a' \wedge b \sim_B b'$$

Multiplicative Disjunction

$$A + B := \mathbf{Coh}\left(\mathcal{A}_A \times \mathcal{A}_B, \sim_{A+B}\right) \text{ where}$$
$$\begin{array}{rcl} (a,b) \sim_{A+B} (a',b') & := & (a,b) = (a',b') \vee \\ & & (a \sim_A a' \wedge a \neq a') \vee (\wedge b \sim_B b' \wedge b \neq b') \end{array}$$

Additive Conjunction

$$A \sqcap B := \mathbf{Coh}\left(\mathcal{A}_A \mathring{\cup} \mathcal{A}_B, \sim_{A\sqcap B}\right)$$

where

$$A \mathring{\cup} B := (\{0\} \times A) \cup (\{1\} \times B)$$

and

$$\begin{array}{rcl} (x,a) \sim_{A\sqcap B} (y,b) & := & ((x = y = 0) \wedge a \sim_A b) \vee \\ & & ((x = y = 1) \wedge a \sim_B b) \vee \\ & & (x \neq y) \end{array}$$

Additive Disjunction

$$A \sqcup B := \mathbf{Coh}\left(\mathcal{A}_A \mathring{\cup} \mathcal{A}_B, \sim_{A\sqcup B}\right) \text{ where}$$
$$(x,a) \sim_{A\sqcup B} (y,b) := ((x = y = 0) \wedge a \sim_A b) \vee ((x = y = 1) \wedge a \sim_B b)$$

Linear Implication

$$A \multimap B := \mathbf{Coh}\left(\mathcal{A}_A \times \mathcal{A}_B, \sim_{A\multimap B}\right) \text{ where}$$
$$(a,b) \sim_{A\multimap B} (a',b') := a \sim_A a' \Rightarrow (b \sim_B b' \wedge (b = b' \Rightarrow a = a'))$$

Constants

$\top$ and $\perp$ correspond to $\underline{0}$
$\mathbf{1}$ and $\mathbf{0}$ correpsond to $\underline{1}$

Exponentials

$$!A := \mathbf{Coh}\left(\mathbf{Fincoh}(A), \sim !A\right) \text{ where } a \sim_{!A} a' := a \cup a' \in \mathbf{Coh}(A)$$
$$?A := \mathbf{Coh}\left(\mathbf{Fincoh}(A), \sim !A\right) \text{ where } a \sim_{!A} a' := a = a' \vee a \cup a' \notin \mathbf{Coh}(A)$$

Table B.2: Webs for LL connectives

Appendix C

Reddy's Linear Logic Model of State (LLMS)

This appendix presents the inference rules and coherence space semantics for $\triangleright$ and $\dagger$ from [21].

C.0.1 Inference Rules

Structural rules

$$\frac{\Gamma' \vdash A}{\Gamma \vdash A} \text{ (Ser)} \quad \text{if } |\Gamma| = |\Gamma'| \text{ and } \leq_\Gamma \subseteq \leq_{\Gamma'}$$

Identity rules

$$A \vdash A \text{ (Ax)} \qquad \frac{\Gamma \vdash A \quad \Delta[A] \vdash B}{\Delta[\Gamma] \vdash B} \text{ (Cut)}$$

Multiplicatives

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \star B} \ (\star\text{R}) \qquad \frac{\Gamma[A, B] \vdash C}{\Gamma[A \star B] \vdash C} \ (\star\text{L})$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma; \Delta \vdash A \triangleright B} \ (\triangleright\text{R}) \qquad \frac{\Gamma[A; B] \vdash C}{\Gamma[A \triangleright B] \vdash C} \ (\triangleright\text{L})$$

$$\vdash \mathbf{1} \ (\mathbf{1}\text{R}) \qquad \frac{\Gamma[\epsilon] \vdash C}{\Gamma[\mathbf{1}] \vdash C} \ (\mathbf{1}\text{L})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \ (\multimap\text{R}) \qquad \frac{\Gamma \vdash A \quad \Delta[B] \vdash C}{\Delta[\Gamma, A \multimap B] \vdash C} \ (\multimap\text{L})$$

Additives

$$\Gamma \vdash \top \quad (\top\mathrm{R})$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \sqcap B}\ (\sqcap\mathrm{R}) \qquad \frac{\Gamma[A] \vdash C}{\Gamma[A \sqcap B] \vdash C}\ (\sqcap\mathrm{L}_1)$$

$$\frac{\Gamma[B] \vdash C}{\Gamma[A \sqcap B] \vdash C}\ (\sqcap\mathrm{L}_2)$$

Modalities

$$\frac{\Gamma[\epsilon] \vdash C}{\Gamma[\dagger A] \vdash C}\ (\dagger\mathrm{Weak}) \qquad \frac{\Gamma[A] \vdash C}{\Gamma[\dagger A] \vdash C}\ (\dagger\mathrm{Der}) \qquad \frac{\Gamma[\dagger A; \dagger A] \vdash C}{\Gamma[\dagger A] \vdash C}\ (\dagger\mathrm{Thread})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \dagger A}\ (\dagger\mathrm{R}) \quad \text{if } \Gamma \text{ is an independent context with only ! or } \dagger \text{ formulas}$$

$$\frac{\Gamma[\dagger A] \vdash C}{\Gamma[!A] \vdash C}\ (!\mathrm{Ser}) \qquad \frac{\Gamma[!A, !A] \vdash C}{\Gamma[!A] \vdash C}\ (!\mathrm{Contr})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash !A}\ (!\mathrm{R}) \quad \text{if } \Gamma \text{ is an independent context with only ! formulas}$$

A natural-deduction-style sequent calculus is obtained by treating the "R" rules as introduction rules, and combining the "L" rules with Cut to derive elimination rules. For example, the $\star$E rule is derived using Cut and $\star$R as follows:

$$\frac{\Gamma \vdash A \star B \qquad \dfrac{\Delta[A, B] \vdash C}{\Delta[A \star B] \vdash C}}{\Delta[\Gamma] \vdash C}$$

The reading of the "L" rules is sometimes counter-intuitive, because they frequently say that the opposite transformations that are allowed on the right of the turnstile are allowed on the left. For example, in †Ser, the rule says that if C can be obtained via a context which assumes only a $\dagger A$, then it could also be obtained from the same context, except with a $!A$. This is because the $!A$ can derive all the same things as the $\dagger A$. The rule does not say that a $\dagger A$ on the right side of a turnstile can be converted into a $!A$. There is a distinction between assumptions and conclusions that must be kept in mind.

In Chapter 3 we say that the Ser rule can be viewed as a pair of rules

$$\frac{\Gamma'[A; B] \vdash C}{\Gamma[A, B] \vdash C} \qquad \frac{\Gamma'[A, B] \vdash C}{\Gamma[B, A] \vdash C}$$

This is because the original rule stipulates that the new context Γ contains the same assumptions as Γ' but the partial order in Γ is a sub-order of the partial order in Γ'. The partial order is determined completely by the distribution of semicolons. Permuting assumptions separated only by commas does not change the order at all, and so produces a (improper) sub-order of the original order. Replacing semicolons with commas produces a strict sub-order of the original partial order. These are the only two ways to change the context that do not discard assumptions or contradict the original order.

The $\star$Eser$_i$ rules are derived using these rules and the $\star$E rule as follows:

$$\dfrac{\Gamma \vdash A \star B \quad \dfrac{\Delta[A;B] \vdash C}{\Delta[A,B] \vdash C}}{\Delta[\Gamma] \vdash C} \qquad \dfrac{\Gamma \vdash B \star A \quad \dfrac{\Delta[A,B] \vdash C}{\Delta[B,A] \vdash C}}{\Delta[\Gamma] \vdash C}$$

These rules say that $A \star B$ can be used in places where a $A \rhd B$ is assumed, and that the order of the components of $A \star B$ does not matter when applying $\star$E. This first claim may seem suspect, but as assumptions A and B have no elaborated structure and so the semicolon between them is there in case A *must* come before B. A completely independent pair $A \star B$ would do just as well. A dependent pair $A \rhd B$ cannot, of course, be plugged into $\Delta[A, B]$, because the comma indicates that A and B are assumed independent.

The new pair $A \rhd B$ can be viewed intuitively as a dependent pair in which B is influenced by A. In contrast the strict pair $A \star B$ is completely independent. The modality $\dagger$ has much the same relationship to $\rhd$ as ! does to $\star$. A $!A$ can be thought of as factory that can produce any number of independent copies of A. A $\dagger A$ can be thought of as a special factory that produces objects which cannot be used independently, but must be used in a sequence. So, whereas

$$!A \vdash\ !A \star !A,$$

we have

$$\dagger A \vdash \dagger A \rhd \dagger A.$$

Moreover, since independent objects may be used in sequences too, everywhere a $\dagger A$ is required, a $!A$ can be used. As is explained below, while the coherence-space interpretation of $!A$ is one of finite coherent sets, the interpretation of $\dagger A$ is one of sequences with special coherence restrictions.

C.0.2 Coherence Space Semantics

The webs for $\rhd$ and $\dagger$ are shown in Table C.1, where $\mathcal{A}_A*$ is the set of finite sequences over $\mathcal{A}_A$. We do not actually make use of the webs, but they are included for completeness.

Before

$$\begin{array}{lcl} \multicolumn{3}{l}{A \rhd B := \mathbf{Coh}\,(\mathcal{A}_A \times \mathcal{A}_B, \sim_{A \rhd B}) \text{ where}} \\ (a,b) \sim_{A\rhd B} (a',b') & := & ((a \sim_A a') \wedge (a \neq a')) \vee \\ & & ((a = a') \wedge (b \sim_B b')) \end{array}$$

Regerative Modality

$$\begin{array}{lcl} \multicolumn{3}{l}{\dagger A := \mathbf{Coh}\,(\mathcal{A}_A*, \sim_{\dagger A}) \text{ where}} \\ a \sim_{\dagger A} a' & := & (a \text{ is a prefix of } a') \vee \\ & & (a' \text{ is a prefix of } a) \vee \\ & & (a = a_0 \cdot \langle \alpha \rangle \cdot s \text{ and } a' = a_0 \cdot \langle \alpha' \rangle \cdot t \text{ and } \alpha \sim_A \alpha' \\ & & \quad \text{for some } a_0, s, t \in \mathcal{A}_A* \text{ and } \alpha, \alpha' \in \mathcal{A}_A) \end{array}$$

Table C.1: Webs for $\rhd$ and $\dagger$.

$\rhd$ is associative and has $\mathbf{1}$ as both left and right unit; it is not commutative. It distributes over both $\sqcap$ and $\sqcup$ on the left,

$$(A \sqcap B) \rhd C \simeq (A \rhd C) \sqcap (B \rhd C)$$
$$(A \sqcup B) \rhd C \simeq (A \rhd C) \sqcup (B \rhd C)$$

and the implications

$$\begin{array}{rcl} A \star (B \triangleright C) & \multimap & (A \star B) \triangleright C \\ (A \triangleright B) \star C & \multimap & (A \triangleright (B \star C) \\ A + (B \triangleright C) & \multimap & (A + B) \triangleright C \\ (A \triangleright B) + C & \multimap & (A \triangleright (B + C) \end{array}$$

can be confirmed. In a categorical semantics, $(\triangleright, \mathbf{1})$ forms a monoid, with $(\star, \mathbf{1})$ as a commutative submonoid, with corresponding linear isomorphisms and injections for the above properties. $A \triangleright B$ is discrete (resp. single-atom) when A and B are discrete (resp. single-atom). Notice that if $\sim_A$ is equality, then $\sim_{A \triangleright B}$ is the same as $\sim_{A \star B}$.

It can be verified that there are linear maps

$$\begin{array}{rcl} \mathbf{done} & : & \dagger A \multimap \mathbf{1} \\ \mathbf{seq} & : & \dagger A \multimap \dagger A \triangleright \dagger A \\ \mathbf{sread} & : & \dagger A \multimap A \\ \mathbf{Seq} & : & \dagger A \multimap \dagger \dagger A \end{array}$$

and a linear injection **Ser** : $!A \multimap \dagger A$ which together justify the inference rules †Ew, †E, †Et, and †I. Like $!A$, $\dagger A$ is never discrete or single-atom.

Appendix D

Full Text of the Proof of the Exchange Function

The full MIZAR-I text of the proof of the `exch` function appears below. Comments appear frequently and elaborate the general structure discussed in Chapter 5.

```
module Exch;

<*
This module derives an exchange operation on arrays of naturals.
That is, an operation that takes an array <a> of length <n> and two
indices <i> and <j> together with their guards and exchanges
the <i>th and <j>th components of <a>.

The procedure for this is obviously trivial:

*       proc Exch (a: Arrayname[N,n], i: N, j: N)
*          var temp: N;
*          temp := a[i];
*          a[i] := a[j];
*          a[j] := temp;
*       end

but in the derivation this breaks up into six basic steps:

Step 1: read a[i]

Step 2: put the value into a variable, temp

Step 3: read a[j]

Step 4: put the value into a[i]

Step 5: read temp

Step 6: put temp into a[j] *>

begin

<* the array dimension *>
```

```
n by assumption ( n:  N );
/* n |- n: N */

<* the array itself *>

a by assumption ( a:  Arrayname[N, n] );
/* a |- a: Arrayname[N, n] */

<* index 1 -- must be reusable, so bang it *>

i by assumption ( i:  !(N) );
/* i |- i: !(N) */

i_ub by bange ( i );
/* i |- i_ub: N */

<* proof that <i> is in bounds -- again must be reusable *>

g by assumption ( g:  !(Get(i) < n) );
/* g |- g: !(Get(i) < n) */

g_ub by bange ( g );
/* g |- g_ub: Get(i) < n */

<* index 2 -- must be reusable, so bang it *>

j by assumption ( j:  !(N) );
/* j |- j: !(N) */

j_ub by bange ( j );
/* j |- j_ub: N */

<* proof that <j> is in bounds -- again must be reusable *>

h by assumption ( h:  !(Get(j) < n) );
/* h |- h: !(Get(j) < n) */

h_ub by bange ( h );
/* h |- h_ub: Get(j) < n */

<*
*
* the operation; we introduce a new block so we can make the
* temporary variable a local variable in the derived function
* *>

op: now

  <* the temporary variable *>

  temp by assumption ( temp:  Varname[N] );
  /* temp |- temp: Varname[N] */
```

```
<*
*
* STEP 1: fetch a[i]
* Notice that information about the meaning of a[i] and the
* effect of the operation on the array is included.
* *>

STEP1 by aget ( a, i_ub, g_ub );
/*
   a, i, g
   |-
   STEP1:
     ex __x232: N .
       !(__x232 = @[a', Get(i), Get(g)]) *
         !(for __x233: N .
             for __x234: __x233 < n .
               (@[a, __x233, __x234] = @[a', __x233, __x234]))
   */

<*
*
* STEP 2: put it into the temporary
* We also need to massage the spec information
* that tags along with the value at a[i]
* Since the array is unchanged at this point, the result should look
* something like
*
*      !(@(temp) = @[a', Get(i), Get(g)])
*      *
*      !(for i1: N. for g1: i1 < n.
*            @[a, i1,g1] = @[a',i1,g1])
*
* ie. new temp = old a[i], new a[i1] = old a[i1] for all i1
* *>

STEP2: now

  <* dummy for the quantified object  from STEP 1 *>

  dSTEP1_ob by assumption ( dSTEP1_ob:  N );
  /* dSTEP1_ob |- dSTEP1_ob: N */

  <* dummy for the spec part from STEP 1 *>

  dSTEP1_spec by assumption (
    dSTEP1_spec:
      (!(dSTEP1_ob = @[a', Get(i), Get(g)]) *
         !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1]))
    );
  /*
     dSTEP1_spec
     |-
     dSTEP1_spec:
       (!(dSTEP1_ob = @[a', Get(i), Get(g)]) *
```

```
        !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1]))
   */

<* assign the dummy value to the temp variable; we will eliminate it
   later
*>

dSTEP1_ob_in_temp by vput ( dSTEP1_ob, temp );
/* dSTEP1_ob, temp |- dSTEP1_ob_in_temp: !(@(temp) = dSTEP1_ob) */

<*
now we need to split up the spec part so we can eventually get
the @(temp) = @[a', Get(i), Get(g)]; dSTEP1_spec_2
is the second part of the spec, which will remain  unchanged *>

dSTEP1_spec_1 by assumption (
  dSTEP1_spec_1:  !(dSTEP1_ob = @[a', Get(i), Get(g)]) );
/* dSTEP1_spec_1 |- dSTEP1_spec_1: !(dSTEP1_ob = @[a', Get(i), Get(g)])
   */

dSTEP1_spec_2 by assumption (
  dSTEP1_spec_2:
    !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1]) );
/*
   dSTEP1_spec_2
   |-
   dSTEP1_spec_2:
     !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1])
   */

<*
we'd like to get a bang on the predicate
@(temp) = @[a',Get(i),Get(g)], and to
do this we need yet another block *>

banged_eq: now

  <*
  dSTEP1_spec_1 is already banged; we need dSTEP1_ob_in_temp but
  with a banged context *>

  ddSTEP1_ob_in_temp by assumption (
    ddSTEP1_ob_in_temp:  !(@(temp) = dSTEP1_ob) );
  /* ddSTEP1_ob_in_temp |- ddSTEP1_ob_in_temp: !(@(temp) = dSTEP1_ob) */

  <* unbanged version of dSTEP1_spec_1; we need to get at the equality *>

  dSTEP1_spec_1_ub by bange ( dSTEP1_spec_1 );
  /* dSTEP1_spec_1 |- dSTEP1_spec_1_ub: dSTEP1_ob = @[a', Get(i), Get(g)]
     */

  <* unbanged version of x3_in_temp; we need to get at the equality *>

  x3_in_temp_ub by bange ( ddSTEP1_ob_in_temp );
```

```
  /* ddSTEP1_ob_in_temp |- x3_in_temp_ub: @(temp) = dSTEP1_ob */

  <* we've got the equality *>

  ai_eq_temp by eqe ( dSTEP1_spec_1_ub, term __dummy,
    type @(temp) = __dummy, x3_in_temp_ub );
  /*
     dSTEP1_spec_1, ddSTEP1_ob_in_temp
     |-
     ai_eq_temp: @(temp) = @[a', Get(i), Get(g)] */

  <* now bang it *>

  b_ai_eq_temp by bangi ( ai_eq_temp );
  /*
     dSTEP1_spec_1, ddSTEP1_ob_in_temp
     |-
     b_ai_eq_temp: !@(temp) = @[a', Get(i), Get(g)] */

  <*
  and eliminate using the Cut rule (or equivalently a function
  application). This last sequent is the result of the block *>

  banged_eq_result by { cut, select} ( dSTEP1_ob_in_temp,
    term ddSTEP1_ob_in_temp, b_ai_eq_temp, num 1 );
  /*
     dSTEP1_spec_1, dSTEP1_ob, temp
     |-
     banged_eq_result: !@(temp) = @[a', Get(i), Get(g)] */

end /*banged_eq*/;
/*
   dSTEP1_spec_1, dSTEP1_ob, temp
   |-
   banged_eq: !@(temp) = @[a', Get(i), Get(g)] */

<* Now, package up the unchanged part of the spec with this result. *>

dNew_spec by mci ( banged_eq, dSTEP1_spec_2 );
/*
   dSTEP1_spec_1, dSTEP1_ob, temp, dSTEP1_spec_2
   |-
   dNew_spec:
     !@(temp) = @[a', Get(i), Get(g)] *
       !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1])
   */

<*
and eliminate the assumptions dSTEP1_spec and dSTEP1_spec_1 using
dSTEP1_ob; the result of plain *-Elim is ambiguous because of
permutations of the context of dNew_spec, but the results are all
equivalent, so just pick the first one *>

dResult by { mce, select} ( dSTEP1_spec, term dSTEP1_spec_1,
```

```
    term dSTEP1_spec_2, dNew_spec, num 1 );
  /*
     dSTEP1_ob, temp, dSTEP1_spec
     |-
     dResult:
       !@(temp) = @[a', Get(i), Get(g)] *
         !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1])
     */

  <*
  finally, we use existential elimination with the result of
  STEP1 to get rid of dSTEP1_ob and dSTEP1_spec *>

  Result by { exe, select} ( term __dummy,
    type (!(__dummy = @[a', Get(i), Get(g)]) *
           !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1])),
    STEP1, term dSTEP1_ob, term dSTEP1_spec, dResult, num 1 );
  /*
     temp, a, i, g
     |-
     Result:
       !@(temp) = @[a', Get(i), Get(g)] *
         !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1])
     */

end /*STEP2*/;
/*
   temp, a, i, g
   |-
   STEP2:
     !@(temp) = @[a', Get(i), Get(g)] *
       !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1])
   */

<* *
   *
   * STEP 3: read a[j]
   * This is easy, just like STEP 1
   *
   * *>

STEP3 by aget ( a, j_ub, h_ub );
/*
   a, j, h
   |-
   STEP3:
     ex __x261: N .
       !(__x261 = @[a', Get(j), Get(h)]) *
         !(for __x262: N .
             for __x263: __x262 < n .
               (@[a, __x262, __x263] = @[a', __x262, __x263]))
   */

<*
```

```
*
* STEP4: put the value read in STEP3 into a[i]
* Our goal is to get an effect-statement that looks like
*
*  !(@[a, i, g] = @[a'', j, h]) *
*  !(for i1: N. for g1: i1 < n.
*      ((i1 < i | i < i1) -o @[a, i1, g1] = @[a'', i1, g1]))
*
* ie. new a[i] = old a[j]
*     and new a[i1] = old a[i1] for all other i1 <> i
*
* *>

STEP4: now

  <* dummy for the object from step 3 *>

  dSTEP3_ob by assumption ( dSTEP3_ob:  N );
  /* dSTEP3_ob |- dSTEP3_ob: N */

  <* dummy for the spec part *>

  dSTEP3_eff by assumption (
    dSTEP3_eff:
      (!(dSTEP3_ob = @[a', Get(j), Get(h)]) *
         !(for j1: N . for h1: j1 < n . @[a, j1, h1] = @[a', j1, h1]))
    );
  /*
     dSTEP3_eff
     |-
     dSTEP3_eff:
       (!(dSTEP3_ob = @[a', Get(j), Get(h)]) *
          !(for j1: N . for h1: j1 < n . @[a, j1, h1] = @[a', j1, h1]))
     */

  <* write the dummy into the array at position i *>

  assign_eff by asput ( dSTEP3_ob, a, i_ub, g_ub );
  /*
     dSTEP3_ob; a, i, g
     |-
     assign_eff:
       !(@[a, Get(i), Get(g)] = dSTEP3_ob) *
         !(for __x266: N .
             for __x267: __x266 < n .
               (__x266 < Get(i)) | (Get(i) < __x266) -o
                 @[a, __x266, __x267] = @[a', __x266, __x267])
     */

  <*
  as in STEP2, we want to manipulate the effect-statements from
  STEP3 and the assignment above *>

  dSTEP3_eff_1 by assumption (
```

```
  dSTEP3_eff_1:  !(dSTEP3_ob = @[a', Get(j), Get(h)]) );
/* dSTEP3_eff_1 |- dSTEP3_eff_1: !(dSTEP3_ob = @[a', Get(j), Get(h)]) */

dSTEP3_eff_1_ub by bange ( dSTEP3_eff_1 );
/* dSTEP3_eff_1 |- dSTEP3_eff_1_ub: dSTEP3_ob = @[a', Get(j), Get(h)] */

dSTEP3_eff_2 by assumption (
  dSTEP3_eff_2:
    !(for j1: N . for h1: j1 < n . @[a, j1, h1] = @[a', j1, h1]) );
/*
   dSTEP3_eff_2
   |-
   dSTEP3_eff_2:
     !(for j1: N . for h1: j1 < n . @[a, j1, h1] = @[a', j1, h1])
   */

dSTEP3_eff_2_ub by bange ( dSTEP3_eff_2 );
/*
   dSTEP3_eff_2
   |-
   dSTEP3_eff_2_ub:
     for j1: N . for h1: j1 < n . @[a, j1, h1] = @[a', j1, h1]
   */

assign_eff_1 by assumption (
  assign_eff_1:  !(@[a, Get(i), Get(g)] = dSTEP3_ob) );
/* assign_eff_1 |- assign_eff_1: !(@[a, Get(i), Get(g)] = dSTEP3_ob) */

assign_eff_1_ub by bange ( assign_eff_1 );
/* assign_eff_1 |- assign_eff_1_ub: @[a, Get(i), Get(g)] = dSTEP3_ob */

assign_eff_2 by assumption (
  assign_eff_2:
    !(for i1: N .
        for g1: i1 < n .
          ((i1 < Get(i) | Get(i) < i1) -o @[a, i1, g1] = @[a', i1, g1]))
  );
/*
   assign_eff_2
   |-
   assign_eff_2:
     !(for i1: N .
         for g1: i1 < n .
           ((i1 < Get(i) | Get(i) < i1) -o @[a, i1, g1] = @[a', i1, g1]))
   */

assign_eff_2_ub by bange ( assign_eff_2 );
/*
   assign_eff_2
   |-
   assign_eff_2_ub:
     for i1: N .
       for g1: i1 < n .
         ((i1 < Get(i) | Get(i) < i1) -o @[a, i1, g1] = @[a', i1, g1])
```

```
   */

<*
to keep things organized, we'll finish massaging the effect-statement
in a new block *>

new_effect: now

  <* first we need a fresh reusable index and guard *>

  i2 by assumption ( i2:  !(N) );
  /* i2 |- i2: !(N) */

  i2_ub by bange ( i2 );
  /* i2 |- i2_ub: N */

  g2 by assumption ( g2:  !(Get(i2) < n) );
  /* g2 |- g2: !(Get(i2) < n) */

  g2_ub by bange ( g2 );
  /* g2 |- g2_ub: Get(i2) < n */

  <* now use these to eliminate the universals *>

  dSTEP3_eff_2_part by fore ( i2_ub, dSTEP3_eff_2_ub );
  /*
     i2, dSTEP3_eff_2
     |-
     dSTEP3_eff_2_part:
       for h1: Get(i2) < n . @[a, Get(i2), h1] = @[a', Get(i2), h1]
     */

  dSTEP3_eff_2_bare by fore ( g2_ub, dSTEP3_eff_2_part );
  /*
     g2, i2, dSTEP3_eff_2
     |-
     dSTEP3_eff_2_bare: @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)]
     */

  <*
  we need to use the sequential versions of universal
  elimination for the second to get extra semicolons; this
  way we can deliberately thread the context later collapsing
  the copies of i2 and g2 and eliminating them with universal
  introduction *>

  assign_eff_2_part1 by fore ( i2_ub, assign_eff_2_ub );
  /*
     i2, assign_eff_2
     |-
     assign_eff_2_part1:
       for g1: Get(i2) < n .
         ((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
            @[a, Get(i2), g1] = @[a', Get(i2), g1])
```

```
   */

assign_eff_2_bare by fore ( g2_ub, assign_eff_2_part1 );
/*
   g2, i2, assign_eff_2
   |-
   assign_eff_2_bare:
     (Get(i2) < Get(i) | Get(i) < Get(i2)) -o
       @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)]
   */

<*
* now, the stuff from STEP2 refers to the previous state of
* the array, prior to the assignment, so we will not be able
* to correctly use it without using >>-Intro.
*
* First, reassemble the stripped-down conjunctions for STEP3
* and the assignment.
* *>

STEP3_eff_simple by mci ( dSTEP3_eff_1_ub, dSTEP3_eff_2_bare );
/*
   dSTEP3_eff_1, g2, i2, dSTEP3_eff_2
   |-
   STEP3_eff_simple:
     dSTEP3_ob = @[a', Get(j), Get(h)] *
       @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)]
   */

<*
need to make sure all the assignment stuff ends up to the right
of the rightmost semicolon *>

assign_eff_simple by mci ( assign_eff_2_bare, assign_eff_1_ub );
/*
   g2, i2, assign_eff_2, assign_eff_1
   |-
   assign_eff_simple:
     (Get(i2) < Get(i) | Get(i) < Get(i2)) -o
       @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)] *
       @[a, Get(i), Get(g)] = dSTEP3_ob
   */

<*
* These types are discrete. We can now combine them with
* >>-Intro and in so doing prime the left side. Then apply
* >>To* (unsafe) to get the prime on the left.
* Then deliberately thread the context to collapse the
* copies of i2 and g2. Since we used foreS above, we have
* enough extra semicolons.
* *>

new_eff_1 by befi ( STEP3_eff_simple, assign_eff_simple );
/*
```

```
   dSTEP3_eff_1, g2, i2, dSTEP3_eff_2; g2, i2, assign_eff_2,
     assign_eff_1
   |-
   new_eff_1:
     dSTEP3_ob = @[a', Get(j), Get(h)] *
       @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)] >>
       (Get(i2) < Get(i) | Get(i) < Get(i2)) -o
         @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)] *
         @[a, Get(i), Get(g)] = dSTEP3_ob
   */

new_eff_2 by { bef2mc, threading} ( new_eff_1 );
/*
   dSTEP3_eff_1, g2, i2, dSTEP3_eff_2, assign_eff_2, assign_eff_1
   |-
   new_eff_2:
     dSTEP3_ob = @[a'', Get(j), Get(h)] *
       @[a', Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)] *
       (Get(i2) < Get(i) | Get(i) < Get(i2)) -o
         @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)] *
         @[a, Get(i), Get(g)] = dSTEP3_ob
   */

<*
* Now we break this down into component assumptions and
* combine them. We'll use another block for organizational
* purposes *>

new_eff_3: now

  part1 by assumption (
    part1:
      (dSTEP3_ob = @[a'', Get(j), Get(h)] *
         @[a', Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)])
    );
  /*
     part1
     |-
     part1:
       (dSTEP3_ob = @[a'', Get(j), Get(h)] *
          @[a', Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)])
     */

  part1_1 by assumption ( part1_1:  dSTEP3_ob = @[a'', Get(j), Get(h)]
    );
  /* part1_1 |- part1_1: dSTEP3_ob = @[a'', Get(j), Get(h)] */

  part1_2 by assumption (
    part1_2:  @[a', Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)] );
  /*
     part1_2
     |-
     part1_2: @[a', Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)] */
```

```
part2 by assumption (
  part2:
    (((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
        @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)])
       * @[a, Get(i), Get(g)] = dSTEP3_ob)
  );
/*
   part2
   |-
   part2:
     (((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
         @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)])
        * @[a, Get(i), Get(g)] = dSTEP3_ob)
   */

part2_1 by assumption ( part2_1:  @[a, Get(i), Get(g)] = dSTEP3_ob );
/* part2_1 |- part2_1: @[a, Get(i), Get(g)] = dSTEP3_ob */

part2_2 by assumption (
  part2_2:
    ((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
       @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)])
  );
/*
   part2_2
   |-
   part2_2:
     ((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
        @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)])
   */

<*
combine the first parts using equality Elim to get rid
of the dummy dSTEP3_ob *>

combine1 by eqe ( part1_1, term __dummy,
  type @[a, Get(i), Get(g)] = __dummy, part2_1 );
/*
   part1_1, part2_1
   |-
   combine1: @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] */

<*
* combine the second part; first assume that i2 is not i,
* and eliminate the implication *>

i2_not_i by assumption (
  i2_not_i:  (Get(i2) < Get(i) | Get(i) < Get(i2)) );
/* i2_not_i |- i2_not_i: (Get(i2) < Get(i) | Get(i) < Get(i2)) */

part2_2_bare by limpe ( i2_not_i, part2_2 );
/*
   i2_not_i, part2_2
   |-
```

```
   part2_2_bare: @[a, Get(i2), Get(g2)] = @[a', Get(i2), Get(g2)] */

<*
now use equality elimination with part1_2 to get rid of
a', leaving only a (after) and a'' (before) *>

combine2 by eqe ( part1_2, term __dummy,
  type @[a, Get(i2), Get(g2)] = __dummy, part2_2_bare );
/*
   part1_2, i2_not_i, part2_2
   |-
   combine2: @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)] */

<* and now abstract away the assumption to get a new
   implication *>

combine2_abs by limpi ( term i2_not_i, combine2 );
/*
   part1_2, part2_2
   |-
   combine2_abs:
     Get(i2) < Get(i) | Get(i) < Get(i2) -o
       @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
   */

<* and now combine them to get the target conjunction *>

combine3 by mci ( combine1, combine2_abs );
/*
   part1_1, part2_1, part1_2, part2_2
   |-
   combine3:
     @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
       Get(i2) < Get(i) | Get(i) < Get(i2) -o
         @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
   */

<* eliminate part1_1 and part1_2 with part1 *>

combine4 by { mce, select} ( part1, term part1_1, term part1_2,
  combine3, num 1 );
/*
   part2_1, part2_2, part1
   |-
   combine4:
     @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
       Get(i2) < Get(i) | Get(i) < Get(i2) -o
         @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
   */

<* eliminate part2_1 and part2_2 with part_1 *>

combine5 by { mce, select} ( part2, term part2_2, term part2_1,
  combine4, num 1 );
```

```
    /*
       part1, part2
       |-
       combine5:
         @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
           Get(i2) < Get(i) | Get(i) < Get(i2) -o
             @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
       */

    <* finally, eliminate part1 and part2 with new_eff2 *>

    result by mce ( new_eff_2, term part1, term part2, combine5 );
    /*
       dSTEP3_eff_1, g2, i2, dSTEP3_eff_2, assign_eff_2, assign_eff_1
       |-
       result:
         @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
           Get(i2) < Get(i) | Get(i) < Get(i2) -o
             @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
       */

  end /*new_eff_3*/;
  /*
     dSTEP3_eff_1, g2, i2, dSTEP3_eff_2, assign_eff_2, assign_eff_1
     |-
     new_eff_3:
       @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
         Get(i2) < Get(i) | Get(i) < Get(i2) -o
           @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
     */

  <*
  now abstract the quantifiers; we'll get a quantifier-independent
  part inside but it doesn't matter for this derivation *>

  new_eff_4 by fori ( term g2, new_eff_3 );
  /*
     dSTEP3_eff_1, i2, dSTEP3_eff_2, assign_eff_2, assign_eff_1
     |-
     new_eff_4:
       (for g2: !(Get(i2) < n) .
          @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
            Get(i2) < Get(i) | Get(i) < Get(i2) -o
              @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)])
     */

  new_eff_5 by fori ( term i2, new_eff_4 );
  /*
     dSTEP3_eff_1, dSTEP3_eff_2, assign_eff_2, assign_eff_1
     |-
     new_eff_5:
       (for i2: !(N) .
          for g2: !(Get(i2) < n) .
            @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
```

```
                Get(i2) < Get(i) | Get(i) < Get(i2) -o
                  @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)])
      */

   <* add a bang so its reusable in the future *>

   new_eff_6 by bangi ( new_eff_5 );
   /*
      dSTEP3_eff_1, dSTEP3_eff_2, assign_eff_2, assign_eff_1
      |-
      new_eff_6:
        !for i2: !(N) .
           for g2: !(Get(i2) < n) .
             @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
               Get(i2) < Get(i) | Get(i) < Get(i2) -o
                 @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
      */

end /*new_effect*/;
/*
   dSTEP3_eff_1, dSTEP3_eff_2, assign_eff_2, assign_eff_1
   |-
   new_effect:
     !for i2: !(N) .
        for g2: !(Get(i2) < n) .
          @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
            Get(i2) < Get(i) | Get(i) < Get(i2) -o
              @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
   */

<* now eliminate assign_eff_1 and assign_eff_2 with *-Elim *>

elim_assign_eff by { mce, select} ( assign_eff, term assign_eff_1,
  term assign_eff_2, new_effect, num 1 );
/*
   dSTEP3_eff_1, dSTEP3_eff_2, dSTEP3_ob; a, i, g
   |-
   elim_assign_eff:
     !for i2: !(N) .
        for g2: !(Get(i2) < n) .
          @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
            Get(i2) < Get(i) | Get(i) < Get(i2) -o
              @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
   */

<* and eliminate dSTEP3_eff_1 and dSTEP_3_eff_2 likewise *>

elim_dSTEP3_effs by { mce, select} ( dSTEP3_eff, term dSTEP3_eff_1,
  term dSTEP3_eff_2, elim_assign_eff, num 1 );
/*
   dSTEP3_ob, dSTEP3_eff; a, i, g
   |-
   elim_dSTEP3_effs:
     !for i2: !(N) .
```

```
           for g2: !(Get(i2) < n) .
             @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
               Get(i2) < Get(i) | Get(i) < Get(i2) -o
                 @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
     */

   <*
   and finally eliminate the STEP3 dummies with STEP3; we
   have to use one of the existential semicolon rules *>

   result by { exe, threading} ( term __dummy,
     type (!(__dummy = @[a', Get(j), Get(h)]) *
             !(for j1: N . for h1: j1 < n . @[a, j1, h1] = @[a', j1, h1])),
     STEP3, term dSTEP3_ob, term dSTEP3_eff, elim_dSTEP3_effs );
   /*
      a, j, h, i, g
      |-
      result:
        !for i2: !(N) .
           for g2: !(Get(i2) < n) .
             @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
               Get(i2) < Get(i) | Get(i) < Get(i2) -o
                 @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
      */

end /*STEP4*/;
/*
   a, j, h, i, g
   |-
   STEP4:
     !for i2: !(N) .
        for g2: !(Get(i2) < n) .
          @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
            Get(i2) < Get(i) | Get(i) < Get(i2) -o
              @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]
   */

<* * STEP 5:
   *
   * again, an easy one. Read the contents of the variable.
   *
   * *>

STEP5 by vget ( temp );
/* temp |- STEP5: ex v: N . !(v = @(temp')) * !(@(temp) = @(temp')) */

<*
* STEP 6: write the value of the variable into a[j]. We use
* array sequential-put to get an extra semicolon that we will
* need later when we thread things together
* *>

STEP6: now
```

```
<* dummy for the object from step 5 *>

dSTEP5_ob by assumption ( dSTEP5_ob:  N );
/* dSTEP5_ob |- dSTEP5_ob: N */

<* dummy for the spec part *>

dSTEP5_eff by assumption (
  dSTEP5_eff:  (!(dSTEP5_ob = @(temp')) * !(@(temp) = @(temp'))) );
/*
   dSTEP5_eff
   |-
   dSTEP5_eff: (!(dSTEP5_ob = @(temp')) * !(@(temp) = @(temp'))) */

<* write the dummy into the array at position j *>

assign_eff by asput ( dSTEP5_ob, a, j_ub, h_ub );
/*
   dSTEP5_ob; a, j, h
   |-
   assign_eff:
     !(@[a, Get(j), Get(h)] = dSTEP5_ob) *
       !(for __x300: N .
           for __x301: __x300 < n .
             (__x300 < Get(j)) | (Get(j) < __x300) -o
               @[a, __x300, __x301] = @[a', __x300, __x301])
   */

<*
now we want to combine the effects to get something like
*
*     !(@[a, j, h] = @(temp')) *
*     !(for j1: N .
*         for h1: j1 < n .
*            ((j1 < j | j < j1) -o @[a, j1, h1] = @[a', j1, h1]))
*
* we don't need the post-op value of temp, so we can forget about
* it. However, the derivation is not as simple as it looks. Unless
* we get a semicolon between temp and a in the result for this
* step, we won't be able to thread the final context. So we have
* to once again unpackage everything, use >>-Intro and >>-To-*,
* fiddle around (though not as much), and repackage everything
* and eliminate dummies.
* *>

dSTEP5_eff_1 by assumption ( dSTEP5_eff_1:  !(dSTEP5_ob = @(temp')) );
/* dSTEP5_eff_1 |- dSTEP5_eff_1: !(dSTEP5_ob = @(temp')) */

dSTEP5_eff_1_ub by bange ( dSTEP5_eff_1 );
/* dSTEP5_eff_1 |- dSTEP5_eff_1_ub: dSTEP5_ob = @(temp') */

<* we don't actually need this part *>

dSTEP5_eff_2 by assumption ( dSTEP5_eff_2:  !(@(temp) = @(temp')) );
```

```
/* dSTEP5_eff_2 |- dSTEP5_eff_2: !(@(temp) = @(temp')) */

mkOne_dSTEP5_eff_2 by bangew ( dSTEP5_eff_2 );
/* dSTEP5_eff_2 |- mkOne_dSTEP5_eff_2: One */

dSTEP5_eff_1_ub_only by onee ( mkOne_dSTEP5_eff_2, dSTEP5_eff_1_ub );
/*
   dSTEP5_eff_2, dSTEP5_eff_1
   |-
   dSTEP5_eff_1_ub_only: dSTEP5_ob = @(temp') */

assign_eff_1 by assumption (
  assign_eff_1:  !(@[a, Get(j), Get(h)] = dSTEP5_ob) );
/* assign_eff_1 |- assign_eff_1: !(@[a, Get(j), Get(h)] = dSTEP5_ob) */

assign_eff_1_ub by bange ( assign_eff_1 );
/* assign_eff_1 |- assign_eff_1_ub: @[a, Get(j), Get(h)] = dSTEP5_ob */

assign_eff_2 by assumption (
  assign_eff_2:
    !(for j1: N .
        for h1: j1 < n .
          ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
  );
/*
   assign_eff_2
   |-
   assign_eff_2:
     !(for j1: N .
         for h1: j1 < n .
           ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
   */

<* the second part need not be discrete *>

dassign_eff by mci ( assign_eff_1_ub, assign_eff_2 );
/*
   assign_eff_1, assign_eff_2
   |-
   dassign_eff:
     @[a, Get(j), Get(h)] = dSTEP5_ob *
       !(for j1: N .
           for h1: j1 < n .
             ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
   */

effect_1 by befi ( dSTEP5_eff_1_ub_only, dassign_eff );
/*
   dSTEP5_eff_2, dSTEP5_eff_1; assign_eff_1, assign_eff_2
   |-
   effect_1:
     dSTEP5_ob = @(temp') >>
       @[a, Get(j), Get(h)] = dSTEP5_ob *
         !(for j1: N .
```

```
             for h1: j1 < n .
               ((j1 < Get(j) | Get(j) < j1) -o
                  @[a, j1, h1] = @[a', j1, h1]))
   */

<*
since temp doesn't appear in the second component, it doesn't get
any additional primes; but we have to do this to work with the
contents *>

effect_2 by bef2mc ( effect_1 );
/*
   dSTEP5_eff_2, dSTEP5_eff_1; assign_eff_1, assign_eff_2
   |-
   effect_2:
     dSTEP5_ob = @(temp') *
       @[a, Get(j), Get(h)] = dSTEP5_ob *
         !(for j1: N .
             for h1: j1 < n .
               ((j1 < Get(j) | Get(j) < j1) -o
                  @[a, j1, h1] = @[a', j1, h1]))
   */

<*
tear the result apart, reduce with equality elimination, and
and then use a series of eliminations on the result *>

effect_3: now

  part1 by assumption ( part1:  dSTEP5_ob = @(temp') );
  /* part1 |- part1: dSTEP5_ob = @(temp') */

  part2 by assumption (
    part2:
      (@[a, Get(j), Get(h)] = dSTEP5_ob *
         !(for j1: N .
             for h1: j1 < n .
               ((j1 < Get(j) | Get(j) < j1) -o
                  @[a, j1, h1] = @[a', j1, h1])))
    );
  /*
     part2
     |-
     part2:
       (@[a, Get(j), Get(h)] = dSTEP5_ob *
          !(for j1: N .
              for h1: j1 < n .
                ((j1 < Get(j) | Get(j) < j1) -o
                   @[a, j1, h1] = @[a', j1, h1])))
     */

  part2_1 by assumption ( part2_1:  @[a, Get(j), Get(h)] = dSTEP5_ob );
  /* part2_1 |- part2_1: @[a, Get(j), Get(h)] = dSTEP5_ob */
```

```
part2_2 by assumption (
  part2_2:
    !(for j1: N .
        for h1: j1 < n .
          ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
  );
/*
   part2_2
   |-
   part2_2:
     !(for j1: N .
         for h1: j1 < n .
           ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
   */

new_part1 by eqe ( part1, term __dummy,
  type @[a, Get(j), Get(h)] = __dummy, part2_1 );
/* part1, part2_1 |- new_part1: @[a, Get(j), Get(h)] = @(temp') */

new_parts by mci ( new_part1, part2_2 );
/*
   part1, part2_1, part2_2
   |-
   new_parts:
     @[a, Get(j), Get(h)] = @(temp') *
       !(for j1: N .
           for h1: j1 < n .
             ((j1 < Get(j) | Get(j) < j1) -o
                @[a, j1, h1] = @[a', j1, h1]))
   */

new_part1_2 by { mce, select} ( part2, term part2_1, term part2_2,
  new_parts, num 1 );
/*
   part1, part2
   |-
   new_part1_2:
     @[a, Get(j), Get(h)] = @(temp') *
       !(for j1: N .
           for h1: j1 < n .
             ((j1 < Get(j) | Get(j) < j1) -o
                @[a, j1, h1] = @[a', j1, h1]))
   */

result by mce ( effect_2, term part1, term part2, new_part1_2 );
/*
   dSTEP5_eff_2, dSTEP5_eff_1; assign_eff_1, assign_eff_2
   |-
   result:
     @[a, Get(j), Get(h)] = @(temp') *
       !(for j1: N .
           for h1: j1 < n .
             ((j1 < Get(j) | Get(j) < j1) -o
                @[a, j1, h1] = @[a', j1, h1]))
```

```
     */

end /*effect_3*/;
/*
   dSTEP5_eff_2, dSTEP5_eff_1; assign_eff_1, assign_eff_2
   |-
   effect_3:
     @[a, Get(j), Get(h)] = @(temp') *
       !(for j1: N .
           for h1: j1 < n .
             ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
   */

<*
we left the bang on the universal but we might as well put on the
whole conjunction in case we need to reuse the lone equality *>

effect_4 by bangi ( effect_3 );
/*
   dSTEP5_eff_2, dSTEP5_eff_1; assign_eff_1, assign_eff_2
   |-
   effect_4:
     !@[a, Get(j), Get(h)] = @(temp') *
        !(for j1: N .
            for h1: j1 < n .
              ((j1 < Get(j) | Get(j) < j1) -o
                 @[a, j1, h1] = @[a', j1, h1]))
   */

<* now we can do our eliminations *>

effect_5 by mce ( assign_eff, term assign_eff_1, term assign_eff_2,
  effect_4 );
/*
   dSTEP5_eff_2, dSTEP5_eff_1; dSTEP5_ob; a, j, h
   |-
   effect_5:
     !@[a, Get(j), Get(h)] = @(temp') *
        !(for j1: N .
            for h1: j1 < n .
              ((j1 < Get(j) | Get(j) < j1) -o
                 @[a, j1, h1] = @[a', j1, h1]))
   */

effect_6 by mce ( dSTEP5_eff, term dSTEP5_eff_1, term dSTEP5_eff_2,
  effect_5 );
/*
   dSTEP5_eff; dSTEP5_ob; a, j, h
   |-
   effect_6:
     !@[a, Get(j), Get(h)] = @(temp') *
        !(for j1: N .
            for h1: j1 < n .
              ((j1 < Get(j) | Get(j) < j1) -o
```

```
                      @[a, j1, h1] = @[a', j1, h1]))
      */

  result by exeSer2 ( term __dummy,
    type (!(__dummy = @(temp')) * !(@(temp) = @(temp'))), STEP5,
    term dSTEP5_ob, term dSTEP5_eff, effect_6 );
  /*
      temp; a, j, h
      |-
      result:
        !@[a, Get(j), Get(h)] = @(temp') *
           !(for j1: N .
                 for h1: j1 < n .
                   ((j1 < Get(j) | Get(j) < j1) -o
                      @[a, j1, h1] = @[a', j1, h1]))
      */

end /*STEP6*/;
/*
   temp; a, j, h
   |-
   STEP6:
     !@[a, Get(j), Get(h)] = @(temp') *
        !(for j1: N .
              for h1: j1 < n .
                ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
   */

<*
*  At this point, we combine the results of steps 2, 4, and 6 to obtain
*  the final effect-statement and build the term that performs the
*  three assignments. The results of those three steps are
*
* For STEP 2:
*
*
* temp, a, i, g  |-
*
* STEP2:
*   (!(@(temp) = @[a', Get(i), Get(g)]) *
*     !(for i1: N .
*         for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1]))
*
*
* For STEP 4: (before prime compaction)
*
* a, j, h, i, g  |-
*
* STEP4:
*   !(for i2: !(N) .
*       for g2: !(Get(i2) < n) .
*         (@[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
*           ((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
*             @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)])))
```

```
*
*
* For STEP 6:
*
* temp; a, j, h  |-
*
* STEP6:
*   !((@[a, Get(j), Get(h)] = @(temp') *
*       !(for j1: N .
*           for h1: j1 < n .
*             ((j1 < Get(j) | Get(j) < j1) -o
*               @[a, j1, h1] = @[a', j1, h1]))))
*
*
* We have exactly the right number of semicolons to do the threading
* if we ensure that a semicolon is placed between each context when
* it is joined (we will have to introduce extras for the dummies
* as above in order to avoid losing semicolons because of repeated
* banged terms.
* *>

STEP2_4_6: now

  <* we'll need a fresh index and guard *>

  k by assumption ( k:  !(N) );
  /* k |- k: !(N) */

  k_ub by bange ( k );
  /* k |- k_ub: N */

  f by assumption ( f:  !(Get(k) < n) );
  /* f |- f: !(Get(k) < n) */

  f_ub by bange ( f );
  /* f |- f_ub: Get(k) < n */

  <* start with dummies for STEP2 and STEP4 *>

  dSTEP2_1 by assumption ( dSTEP2_1:  !(@(temp) = @[a', Get(i), Get(g)]) );
  /* dSTEP2_1 |- dSTEP2_1: !(@(temp) = @[a', Get(i), Get(g)]) */

  dSTEP2_1_ub by bange ( dSTEP2_1 );
  /* dSTEP2_1 |- dSTEP2_1_ub: @(temp) = @[a', Get(i), Get(g)] */

  dSTEP2_2 by assumption (
    dSTEP2_2:  !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1])
    );
  /*
     dSTEP2_2
     |-
     dSTEP2_2: !(for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1])
     */
```

```
dSTEP2_2_ub by bange ( dSTEP2_2 );
/*
   dSTEP2_2
   |-
   dSTEP2_2_ub: for i1: N . for g1: i1 < n . @[a, i1, g1] = @[a', i1, g1]
   */

dSTEP4 by assumption (
  dSTEP4:
    !(for i2: !(N) .
        for g2: !(Get(i2) < n) .
          (@[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
             ((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
                @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)])))
  );
/*
   dSTEP4
   |-
   dSTEP4:
     !(for i2: !(N) .
         for g2: !(Get(i2) < n) .
           (@[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
              ((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
                 @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)])))
   */

dSTEP4_ub by bange ( dSTEP4 );
/*
   dSTEP4
   |-
   dSTEP4_ub:
     for i2: !(N) .
       for g2: !(Get(i2) < n) .
         (@[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
            ((Get(i2) < Get(i) | Get(i) < Get(i2)) -o
               @[a, Get(i2), Get(g2)] = @[a'', Get(i2), Get(g2)]))
   */

<*
we will need the universal from STEP 2 instantiated on both the
fresh index k and on j *>

dSTEP2_2_j by fore ( j_ub, dSTEP2_2_ub );
/*
   j, dSTEP2_2
   |-
   dSTEP2_2_j: for g1: Get(j) < n . @[a, Get(j), g1] = @[a', Get(j), g1]
   */

dSTEP2_2_jh by fore ( h_ub, dSTEP2_2_j );
/*
   h, j, dSTEP2_2
   |-
   dSTEP2_2_jh: @[a, Get(j), Get(h)] = @[a', Get(j), Get(h)] */
```

```
dSTEP2_2_k by fore ( k_ub, dSTEP2_2_ub );
/*
   k, dSTEP2_2
   |-
   dSTEP2_2_k: for g1: Get(k) < n . @[a, Get(k), g1] = @[a', Get(k), g1]
   */

dSTEP2_2_kf by fore ( f_ub, dSTEP2_2_k );
/*
   f, k, dSTEP2_2
   |-
   dSTEP2_2_kf: @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)] */

<* assemble the discrete object for in preparation for >>-Intro *>

dSTEP2_disc1 by mci ( dSTEP2_2_jh, dSTEP2_2_kf );
/*
   h, j, dSTEP2_2, f, k, dSTEP2_2
   |-
   dSTEP2_disc1:
     @[a, Get(j), Get(h)] = @[a', Get(j), Get(h)] *
       @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
   */

dSTEP2_disc by mci ( dSTEP2_1_ub, dSTEP2_disc1 );
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k, dSTEP2_2
   |-
   dSTEP2_disc:
     @(temp) = @[a', Get(i), Get(g)] *
       @[a, Get(j), Get(h)] = @[a', Get(j), Get(h)] *
         @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
   */

<* instantiate the universal from STEP 4 on k *>

dSTEP4_k by fore ( k, dSTEP4_ub );
/*
   k, dSTEP4
   |-
   dSTEP4_k:
     for g2: !(Get(k) < n) .
       (@[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
          ((Get(k) < Get(i) | Get(i) < Get(k)) -o
             @[a, Get(k), Get(g2)] = @[a'', Get(k), Get(g2)]))
   */

dSTEP4_kf by fore ( f, dSTEP4_k );
/*
   f, k, dSTEP4
   |-
   dSTEP4_kf:
     @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
```

```
      ((Get(k) < Get(i) | Get(i) < Get(k)) -o
         @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)])
   */

<* we haven't implemented prime compaction yet so fake it with
   an axiom *>

prime_com1 by axiom (
  prime_com1:
    ((@[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
        ((Get(k) < Get(i) | Get(i) < Get(k)) -o
           @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]))
      -o
      (@[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] *
         ((Get(k) < Get(i) | Get(i) < Get(k)) -o
            @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])))
  );
/*

   |-
   prime_com1:
     ((@[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
         ((Get(k) < Get(i) | Get(i) < Get(k)) -o
            @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]))
       -o
       (@[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] *
          ((Get(k) < Get(i) | Get(i) < Get(k)) -o
             @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])))
   */

dSTEP4_kf_pc by limpe ( dSTEP4_kf, prime_com1 );
/*
   f, k, dSTEP4
   |-
   dSTEP4_kf_pc:
     @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] *
       ((Get(k) < Get(i) | Get(i) < Get(k)) -o
          @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<*
now join them with >> to get the shape of the context we want,
and to set up the extra priming *>

effect_1 by befi ( dSTEP2_disc, dSTEP4_kf_pc );
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k, dSTEP2_2; f, k, dSTEP4
   |-
   effect_1:
     @(temp) = @[a', Get(i), Get(g)] *
       @[a, Get(j), Get(h)] = @[a', Get(j), Get(h)] *
         @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
       >>
       @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] *
```

```
        ((Get(k) < Get(i) | Get(i) < Get(k)) -o
           @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<*
and use >>To* to get extra primes on "a" in the first
component; note we have to use the unsafe variety *>

effect_2 by bef2mc_unsafe ( effect_1 );
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k, dSTEP2_2; f, k, dSTEP4
   |-
   effect_2:
     @(temp) = @[a'', Get(i), Get(g)] *
       @[a', Get(j), Get(h)] = @[a'', Get(j), Get(h)] *
         @[a', Get(k), Get(f)] = @[a'', Get(k), Get(f)]
       *
       @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] *
         ((Get(k) < Get(i) | Get(i) < Get(k)) -o
            @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<*
now, as usual, we smash this into assumptions, reduce to some
result, and the eliminate the dummy assumption with the
original *>

effect_3: now

  part1 by assumption (
    part1:
      (@(temp) = @[a'', Get(i), Get(g)] *
         (@[a', Get(j), Get(h)] = @[a'', Get(j), Get(h)] *
            @[a', Get(k), Get(f)] = @[a'', Get(k), Get(f)]))
    );
  /*
     part1
     |-
     part1:
       (@(temp) = @[a'', Get(i), Get(g)] *
          (@[a', Get(j), Get(h)] = @[a'', Get(j), Get(h)] *
             @[a', Get(k), Get(f)] = @[a'', Get(k), Get(f)]))
     */

  part1_1 by assumption ( part1_1:  @(temp) = @[a'', Get(i), Get(g)] );
  /* part1_1 |- part1_1: @(temp) = @[a'', Get(i), Get(g)] */

  part1_2 by assumption (
    part1_2:
      (@[a', Get(j), Get(h)] = @[a'', Get(j), Get(h)] *
         @[a', Get(k), Get(f)] = @[a'', Get(k), Get(f)])
    );
  /*
     part1_2
```

```
   |-
   part1_2:
     (@[a', Get(j), Get(h)] = @[a'', Get(j), Get(h)] *
        @[a', Get(k), Get(f)] = @[a'', Get(k), Get(f)])
   */

part1_2_1 by assumption (
  part1_2_1:  @[a', Get(j), Get(h)] = @[a'', Get(j), Get(h)] );
/*
   part1_2_1
   |-
   part1_2_1: @[a', Get(j), Get(h)] = @[a'', Get(j), Get(h)] */

part1_2_2 by assumption (
  part1_2_2:  @[a', Get(k), Get(f)] = @[a'', Get(k), Get(f)] );
/*
   part1_2_2
   |-
   part1_2_2: @[a', Get(k), Get(f)] = @[a'', Get(k), Get(f)] */

part2 by assumption (
  part2:
    (@[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] *
       ((Get(k) < Get(i) | Get(i) < Get(k)) -o
          @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
  );
/*
   part2
   |-
   part2:
     (@[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] *
        ((Get(k) < Get(i) | Get(i) < Get(k)) -o
           @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
   */

part2_1 by assumption (
  part2_1:  @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] );
/* part2_1 |- part2_1: @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)] */

part2_2 by assumption (
  part2_2:
    ((Get(k) < Get(i) | Get(i) < Get(k)) -o
       @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
  );
/*
   part2_2
   |-
   part2_2:
     ((Get(k) < Get(i) | Get(i) < Get(k)) -o
        @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<* eliminate a'[j] to get a[i] = a''[j] *>
```

```
new_part2 by eqe ( part1_2_1, term __dummy,
  type @[a, Get(i), Get(g)] = __dummy, part2_1 );
/*
   part1_2_1, part2_1
   |-
   new_part2: @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] */

<*
now assume that k <> i, combine the consequent of part2_2
with part1_2_2 to get a[k] = a''[k] *>

new_part3: now

  part2_2_ant by assumption (
    part2_2_ant:  (Get(k) < Get(i) | Get(i) < Get(k)) );
  /* part2_2_ant |- part2_2_ant: (Get(k) < Get(i) | Get(i) < Get(k)) */

  part2_2_cons by limpe ( part2_2_ant, part2_2 );
  /*
     part2_2_ant, part2_2
     |-
     part2_2_cons: @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)] */

  new_part3_cons by eqe ( part1_2_2, term __dummy,
    type @[a, Get(k), Get(f)] = __dummy, part2_2_cons );
  /*
     part1_2_2, part2_2_ant, part2_2
     |-
     new_part3_cons: @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)] */

  result by limpi ( term part2_2_ant, new_part3_cons );
  /*
     part1_2_2, part2_2
     |-
     result:
       Get(k) < Get(i) | Get(i) < Get(k) -o
         @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
     */

end /*new_part3*/;
/*
   part1_2_2, part2_2
   |-
   new_part3:
     Get(k) < Get(i) | Get(i) < Get(k) -o
       @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
   */

<*
put the new parts 2 and 3 together, and then together with
the original part1 *>

new_part2_3 by mci ( new_part2, new_part3 );
/*
```

```
   part1_2_1, part2_1, part1_2_2, part2_2
   |-
   new_part2_3:
     @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
       Get(k) < Get(i) | Get(i) < Get(k) -o
         @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
   */

new_part1_2_3 by mci ( part1_1, new_part2_3 );
/*
   part1_1, part1_2_1, part2_1, part1_2_2, part2_2
   |-
   new_part1_2_3:
     @(temp) = @[a'', Get(i), Get(g)] *
       @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
         Get(k) < Get(i) | Get(i) < Get(k) -o
           @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
   */

<* now replace dummies in the context; this takes a couple of
   *-Elims *>

new_part1_2_3_n1 by { mce, select} ( part2, term part2_1, term
  part2_2, new_part1_2_3, num 1 );
/*
   part1_1, part1_2_1, part1_2_2, part2
   |-
   new_part1_2_3_n1:
     @(temp) = @[a'', Get(i), Get(g)] *
       @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
         Get(k) < Get(i) | Get(i) < Get(k) -o
           @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
   */

new_part1_2_3_n2 by { mce, select} ( part1_2, term part1_2_1,
  term part1_2_2, new_part1_2_3_n1, num 1 );
/*
   part1_1, part2, part1_2
   |-
   new_part1_2_3_n2:
     @(temp) = @[a'', Get(i), Get(g)] *
       @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
         Get(k) < Get(i) | Get(i) < Get(k) -o
           @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
   */

new_part1_2_3_n3 by { mce, select} ( part1, term part1_1, term
  part1_2, new_part1_2_3_n2, num 1 );
/*
   part2, part1
   |-
   new_part1_2_3_n3:
     @(temp) = @[a'', Get(i), Get(g)] *
       @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
```

```
          Get(k) < Get(i) | Get(i) < Get(k) -o
            @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
      */

   result by { mce, contraction} ( effect_2, term part1, term part2,
     new_part1_2_3_n3 );
   /*
      dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4
      |-
      result:
        @(temp) = @[a'', Get(i), Get(g)] *
          @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
            Get(k) < Get(i) | Get(i) < Get(k) -o
              @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
      */

end /*effect_3*/;
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4
   |-
   effect_3:
     @(temp) = @[a'', Get(i), Get(g)] *
       @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
         Get(k) < Get(i) | Get(i) < Get(k) -o
           @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
   */

<*
* We could repackage this, but there is really no point. We might as
* well introduce the dummy for STEP 6, break it down and sequence it
* with effect 3, apply >>-To-*, etc, and only then package up the
* final result *>

dSTEP6 by assumption (
  dSTEP6:
    !(@[a, Get(j), Get(h)] = @(temp') *
        !(for j1: N .
            for h1: j1 < n .
              ((j1 < Get(j) | Get(j) < j1) -o
                 @[a, j1, h1] = @[a', j1, h1]))
  );
/*
   dSTEP6
   |-
   dSTEP6:
     !(@[a, Get(j), Get(h)] = @(temp') *
         !(for j1: N .
             for h1: j1 < n .
               ((j1 < Get(j) | Get(j) < j1) -o
                  @[a, j1, h1] = @[a', j1, h1]))
   */

dSTEP6_ub by bange ( dSTEP6 );
/*
```

```
   dSTEP6
   |-
   dSTEP6_ub:
     @[a, Get(j), Get(h)] = @(temp') *
       !(for j1: N .
           for h1: j1 < n .
             ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
   */

dSTEP6_1 by assumption ( dSTEP6_1:  @[a, Get(j), Get(h)] = @(temp') );
/* dSTEP6_1 |- dSTEP6_1: @[a, Get(j), Get(h)] = @(temp') */

dSTEP6_2 by assumption (
  dSTEP6_2:
    !(for j1: N .
        for h1: j1 < n .
          ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
  );
/*
   dSTEP6_2
   |-
   dSTEP6_2:
     !(for j1: N .
         for h1: j1 < n .
           ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1]))
   */

dSTEP6_2_ub by bange ( dSTEP6_2 );
/*
   dSTEP6_2
   |-
   dSTEP6_2_ub:
     for j1: N .
       for h1: j1 < n .
         ((j1 < Get(j) | Get(j) < j1) -o @[a, j1, h1] = @[a', j1, h1])
   */

<*
* Our target effect-statement is
*
*    @(temp') = @[a''', Get(i), Get(g)] *
*    @[a', Get(i), Get(g)] = @[a''', Get(j), Get(h)] *
*    (Get(k) < Get(i) | Get(i) < Get(k)) -o
*      @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)]
*   *
*    @[a, Get(j), Get(h)] = @(temp') *
*    (Get(i) < Get(j) | Get(i) < Get(k)) -o
*      @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)] *
*    (Get(k) < Get(j) | Get(j) < Get(k)) -o
*      @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
*
* which we can massage in the usual way, eliminating temp'
* and a' in the process. Notice that we have instantiated
* dSTEP6_2_ub above *twice*, once on k,f and once on i,g.
```

```
* Ultimately we have to do a case analysis to deal with the
* possibility that i and j are equal. *>

dSTEP6_2_k by fore ( k_ub, dSTEP6_2_ub );
/*
   k, dSTEP6_2
   |-
   dSTEP6_2_k:
     for h1: Get(k) < n .
       ((Get(k) < Get(j) | Get(j) < Get(k)) -o
          @[a, Get(k), h1] = @[a', Get(k), h1])
   */

dSTEP6_2_kf by fore ( f_ub, dSTEP6_2_k );
/*
   f, k, dSTEP6_2
   |-
   dSTEP6_2_kf:
     (Get(k) < Get(j) | Get(j) < Get(k)) -o
       @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
   */

dSTEP6_2_i by fore ( i_ub, dSTEP6_2_ub );
/*
   i, dSTEP6_2
   |-
   dSTEP6_2_i:
     for h1: Get(i) < n .
       ((Get(i) < Get(j) | Get(j) < Get(i)) -o
          @[a, Get(i), h1] = @[a', Get(i), h1])
   */

dSTEP6_2_ig by fore ( g_ub, dSTEP6_2_i );
/*
   g, i, dSTEP6_2
   |-
   dSTEP6_2_ig:
     (Get(i) < Get(j) | Get(j) < Get(i)) -o
       @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)]
   */

dSTEP6_2_ik by { mci, contraction} ( dSTEP6_2_ig, dSTEP6_2_kf );
/*
   g, i, dSTEP6_2, f, k
   |-
   dSTEP6_2_ik:
     (Get(i) < Get(j) | Get(j) < Get(i)) -o
       @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)] *
       (Get(k) < Get(j) | Get(j) < Get(k)) -o
         @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
   */

new_dSTEP6 by mci ( dSTEP6_1, dSTEP6_2_ik );
/*
```

```
   dSTEP6_1, g, i, dSTEP6_2, f, k
   |-
   new_dSTEP6:
     @[a, Get(j), Get(h)] = @(temp') *
       (Get(i) < Get(j) | Get(j) < Get(i)) -o
         @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)] *
         (Get(k) < Get(j) | Get(j) < Get(k)) -o
           @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
   */

<* now we can apply >>-Intro and >>-To-* *>

effect_4 by befi ( effect_3, new_dSTEP6 );
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4; dSTEP6_1, g, i,
     dSTEP6_2, f, k
   |-
   effect_4:
     @(temp) = @[a'', Get(i), Get(g)] *
       @[a, Get(i), Get(g)] = @[a'', Get(j), Get(h)] *
         Get(k) < Get(i) | Get(i) < Get(k) -o
           @[a, Get(k), Get(f)] = @[a'', Get(k), Get(f)]
       >>
       @[a, Get(j), Get(h)] = @(temp') *
         (Get(i) < Get(j) | Get(j) < Get(i)) -o
           @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)] *
           (Get(k) < Get(j) | Get(j) < Get(k)) -o
             @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
   */

effect_5 by bef2mc_unsafe ( effect_4 );
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4; dSTEP6_1, g, i,
     dSTEP6_2, f, k
   |-
   effect_5:
     @(temp') = @[a''', Get(i), Get(g)] *
       @[a', Get(i), Get(g)] = @[a''', Get(j), Get(h)] *
         Get(k) < Get(i) | Get(i) < Get(k) -o
           @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)]
       *
       @[a, Get(j), Get(h)] = @(temp') *
         (Get(i) < Get(j) | Get(j) < Get(i)) -o
           @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)] *
           (Get(k) < Get(j) | Get(j) < Get(k)) -o
             @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]
   */

<* and start the reduction process *>

effect_6: now

  part1 by assumption (
    part1:
```

```
    (@(temp') = @[a''', Get(i), Get(g)] *
       (@[a', Get(i), Get(g)] = @[a''', Get(j), Get(h)] *
          ((Get(k) < Get(i) | Get(i) < Get(k)) -o
             @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)])))
  );
/*
   part1
   |-
   part1:
     (@(temp') = @[a''', Get(i), Get(g)] *
        (@[a', Get(i), Get(g)] = @[a''', Get(j), Get(h)] *
           ((Get(k) < Get(i) | Get(i) < Get(k)) -o
              @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)])))
   */

part1_1 by assumption ( part1_1:  @(temp') = @[a''', Get(i), Get(g)] );
/* part1_1 |- part1_1: @(temp') = @[a''', Get(i), Get(g)] */

part1_2 by assumption (
  part1_2:
    (@[a', Get(i), Get(g)] = @[a''', Get(j), Get(h)] *
       ((Get(k) < Get(i) | Get(i) < Get(k)) -o
          @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)]))
  );
/*
   part1_2
   |-
   part1_2:
     (@[a', Get(i), Get(g)] = @[a''', Get(j), Get(h)] *
        ((Get(k) < Get(i) | Get(i) < Get(k)) -o
           @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)]))
   */

part1_2_1 by assumption (
  part1_2_1:  @[a', Get(i), Get(g)] = @[a''', Get(j), Get(h)] );
/*
   part1_2_1
   |-
   part1_2_1: @[a', Get(i), Get(g)] = @[a''', Get(j), Get(h)] */

part1_2_2 by assumption (
  part1_2_2:
    ((Get(k) < Get(i) | Get(i) < Get(k)) -o
       @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)])
  );
/*
   part1_2_2
   |-
   part1_2_2:
     ((Get(k) < Get(i) | Get(i) < Get(k)) -o
        @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)])
   */

part2 by assumption (
```

```
  part2:
    (@[a, Get(j), Get(h)] = @(temp') *
       (((Get(i) < Get(j) | Get(j) < Get(i)) -o
           @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)])
          *
          ((Get(k) < Get(j) | Get(j) < Get(k)) -o
             @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])))
  );
/*
   part2
   |-
   part2:
     (@[a, Get(j), Get(h)] = @(temp') *
        (((Get(i) < Get(j) | Get(j) < Get(i)) -o
            @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)])
           *
           ((Get(k) < Get(j) | Get(j) < Get(k)) -o
              @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])))
   */

part2_1 by assumption ( part2_1:  @[a, Get(j), Get(h)] = @(temp') );
/* part2_1 |- part2_1: @[a, Get(j), Get(h)] = @(temp') */

part2_2 by assumption (
  part2_2:
    (((Get(i) < Get(j) | Get(j) < Get(i)) -o
        @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)])
       *
       ((Get(k) < Get(j) | Get(j) < Get(k)) -o
          @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
  );
/*
   part2_2
   |-
   part2_2:
     (((Get(i) < Get(j) | Get(j) < Get(i)) -o
         @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)])
        *
        ((Get(k) < Get(j) | Get(j) < Get(k)) -o
           @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
   */

part2_2_1 by assumption (
  part2_2_1:
    ((Get(i) < Get(j) | Get(j) < Get(i)) -o
       @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)])
  );
/*
   part2_2_1
   |-
   part2_2_1:
     ((Get(i) < Get(j) | Get(j) < Get(i)) -o
        @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)])
   */
```

```
part2_2_2 by assumption (
  part2_2_2:
    ((Get(k) < Get(j) | Get(j) < Get(k)) -o
       @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
  );
/*
   part2_2_2
   |-
   part2_2_2:
     ((Get(k) < Get(j) | Get(j) < Get(k)) -o
        @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<* eliminate temp' to get a[j] = a'''[i] *>

new_part1 by eqe ( part1_1, term __dummy,
  type @[a, Get(j), Get(h)] = __dummy, part2_1 );
/*
   part1_1, part2_1
   |-
   new_part1: @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] */

<*
assume that i <> j and use part2_2_1 and part1_2_1 to conclude
that a[i] = a'''[j]. The case that i=j will be handled after
we get the completely reduced form *>

new_part2: now

  part2_2_1_ant by assumption (
    part2_2_1_ant:  (Get(i) < Get(j) | Get(j) < Get(i)) );
  /*
     part2_2_1_ant
     |-
     part2_2_1_ant: (Get(i) < Get(j) | Get(j) < Get(i)) */

  part2_2_1_cons by limpe ( part2_2_1_ant, part2_2_1 );
  /*
     part2_2_1_ant, part2_2_1
     |-
     part2_2_1_cons: @[a, Get(i), Get(g)] = @[a', Get(i), Get(g)] */

  new_cons by eqe ( part1_2_1, term __dummy,
    type @[a, Get(i), Get(g)] = __dummy, part2_2_1_cons );
  /*
     part1_2_1, part2_2_1_ant, part2_2_1
     |-
     new_cons: @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)] */

  result by limpi ( term part2_2_1_ant, new_cons );
  /*
     part1_2_1, part2_2_1
     |-
```

```
      result:
        Get(i) < Get(j) | Get(j) < Get(i) -o
          @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
      */

end /*new_part2*/;
/*
   part1_2_1, part2_2_1
   |-
   new_part2:
     Get(i) < Get(j) | Get(j) < Get(i) -o
       @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
   */

<*
and now combine part1_2_2 and part2_2_2 to get the case for
k <> i and k <> j *>

new_part3: now

  <* this is just for getting the form we want *>

  k_ne_i_ne_j by assumption (
    k_ne_i_ne_j:
      ((Get(k) < Get(i) | Get(i) < Get(k)) *
         (Get(k) < Get(j) | Get(j) < Get(k)))
    );
  /*
     k_ne_i_ne_j
     |-
     k_ne_i_ne_j:
       ((Get(k) < Get(i) | Get(i) < Get(k)) *
          (Get(k) < Get(j) | Get(j) < Get(k)))
     */

  k_ne_i by assumption ( k_ne_i:  (Get(k) < Get(i) | Get(i) < Get(k))
    );
  /* k_ne_i |- k_ne_i: (Get(k) < Get(i) | Get(i) < Get(k)) */

  k_ne_j by assumption ( k_ne_j:  (Get(k) < Get(j) | Get(j) < Get(k))
    );
  /* k_ne_j |- k_ne_j: (Get(k) < Get(j) | Get(j) < Get(k)) */

  part1_2_2_cons by limpe ( k_ne_i, part1_2_2 );
  /*
     k_ne_i, part1_2_2
     |-
     part1_2_2_cons: @[a', Get(k), Get(f)] = @[a''', Get(k), Get(f)] */

  part2_2_2_cons by limpe ( k_ne_j, part2_2_2 );
  /*
     k_ne_j, part2_2_2
     |-
     part2_2_2_cons: @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)] */
```

```
  new_cons by eqe ( part1_2_2_cons, term __dummy,
    type @[a, Get(k), Get(f)] = __dummy, part2_2_2_cons );
  /*
     k_ne_i, part1_2_2, k_ne_j, part2_2_2
     |-
     new_cons: @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)] */

  new_cons1 by { mce, select} ( k_ne_i_ne_j, term k_ne_i, term
    k_ne_j, new_cons, num 1 );
  /*
     part1_2_2, part2_2_2, k_ne_i_ne_j
     |-
     new_cons1: @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)] */

  result by limpi ( term k_ne_i_ne_j, new_cons1 );
  /*
     part1_2_2, part2_2_2
     |-
     result:
       (Get(k) < Get(i) | Get(i) < Get(k)) *
         (Get(k) < Get(j) | Get(j) < Get(k)) -o
         @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
     */

end /*new_part3*/;
/*
   part1_2_2, part2_2_2
   |-
   new_part3:
     (Get(k) < Get(i) | Get(i) < Get(k)) *
       (Get(k) < Get(j) | Get(j) < Get(k)) -o
       @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
   */

<* assemble the new parts of the effect statement *>

new_part1_2 by mci ( new_part1, new_part2 );
/*
   part1_1, part2_1, part1_2_1, part2_2_1
   |-
   new_part1_2:
     @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
       Get(i) < Get(j) | Get(j) < Get(i) -o
         @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
   */

new_parts by mci ( new_part1_2, new_part3 );
/*
   part1_1, part2_1, part1_2_1, part2_2_1, part1_2_2, part2_2_2
   |-
   new_parts:
     @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
       Get(i) < Get(j) | Get(j) < Get(i) -o
```

```
          @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
        *
        (Get(k) < Get(i) | Get(i) < Get(k)) *
          (Get(k) < Get(j) | Get(j) < Get(k)) -o
          @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
   */

<* now the usual crop of *-Elims to undo a layer of dummy
   assumptions *>

new_parts_n1 by { mce, select} ( part2_2, term part2_2_1,
  term part2_2_2, new_parts, num 1 );
/*
   part1_1, part2_1, part1_2_1, part1_2_2, part2_2
   |-
   new_parts_n1:
     @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
       Get(i) < Get(j) | Get(j) < Get(i) -o
         @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
       *
       (Get(k) < Get(i) | Get(i) < Get(k)) *
         (Get(k) < Get(j) | Get(j) < Get(k)) -o
         @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
   */

new_parts_n2 by { mce, select} ( part2, term part2_1, term part2_2,
  new_parts_n1, num 1 );
/*
   part1_1, part1_2_1, part1_2_2, part2
   |-
   new_parts_n2:
     @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
       Get(i) < Get(j) | Get(j) < Get(i) -o
         @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
       *
       (Get(k) < Get(i) | Get(i) < Get(k)) *
         (Get(k) < Get(j) | Get(j) < Get(k)) -o
         @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
   */

new_parts_n3 by { mce, select} ( part1_2, term part1_2_1,
  term part1_2_2, new_parts_n2, num 1 );
/*
   part1_1, part2, part1_2
   |-
   new_parts_n3:
     @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
       Get(i) < Get(j) | Get(j) < Get(i) -o
         @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
       *
       (Get(k) < Get(i) | Get(i) < Get(k)) *
         (Get(k) < Get(j) | Get(j) < Get(k)) -o
         @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
   */
```

```
  new_parts_n4 by { mce, select} ( part1, term part1_1, term part1_2,
    new_parts_n3, num 1 );
  /*
     part2, part1
     |-
     new_parts_n4:
       @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
         Get(i) < Get(j) | Get(j) < Get(i) -o
           @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
         *
         (Get(k) < Get(i) | Get(i) < Get(k)) *
           (Get(k) < Get(j) | Get(j) < Get(k)) -o
           @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
     */

  result by mce ( effect_5, term part1, term part2, new_parts_n4 );
  /*
     dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4; dSTEP6_1, g, i,
       dSTEP6_2, f, k
     |-
     result:
       @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
         Get(i) < Get(j) | Get(j) < Get(i) -o
           @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
         *
         (Get(k) < Get(i) | Get(i) < Get(k)) *
           (Get(k) < Get(j) | Get(j) < Get(k)) -o
           @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
     */

end /*effect_6*/;
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4; dSTEP6_1, g, i,
     dSTEP6_2, f, k
   |-
   effect_6:
     @[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
       Get(i) < Get(j) | Get(j) < Get(i) -o
         @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]
       *
       (Get(k) < Get(i) | Get(i) < Get(k)) *
         (Get(k) < Get(j) | Get(j) < Get(k)) -o
         @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]
   */

<*
* for the case analysis below, we are going to need to use
* effect-6 twice; if we eliminate dSTEP6_1 and dSTEP6_2 with
* dSTEP6, then everything in the left context would be banged, and
* we could use it twice as it stands. But this would lead to
* concatenating the context with itself, which isn't what we
* want. Thus we'll bang it and then introduce a dummy assumption
* in the case analysis that we can eliminate later.
```

```
*
* First, do the prime-compaction: a''' becomes just a' *>

prime_compact_2 by axiom (
  prime_compact_2:
    (((@[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
         ((Get(i) < Get(j) | Get(j) < Get(i)) -o
            @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]))
        *
        (((Get(k) < Get(i) | Get(i) < Get(k)) *
            (Get(k) < Get(j) | Get(j) < Get(k)))
          -o @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]))
      -o
      ((@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
          ((Get(i) < Get(j) | Get(j) < Get(i)) -o
             @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
         *
         (((Get(k) < Get(i) | Get(i) < Get(k)) *
             (Get(k) < Get(j) | Get(j) < Get(k)))
           -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])))
  );
/*

   |-
   prime_compact_2:
     (((@[a, Get(j), Get(h)] = @[a''', Get(i), Get(g)] *
          ((Get(i) < Get(j) | Get(j) < Get(i)) -o
             @[a, Get(i), Get(g)] = @[a''', Get(j), Get(h)]))
         *
         (((Get(k) < Get(i) | Get(i) < Get(k)) *
             (Get(k) < Get(j) | Get(j) < Get(k)))
           -o @[a, Get(k), Get(f)] = @[a''', Get(k), Get(f)]))
       -o
       ((@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
           ((Get(i) < Get(j) | Get(j) < Get(i)) -o
              @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
          *
          (((Get(k) < Get(i) | Get(i) < Get(k)) *
              (Get(k) < Get(j) | Get(j) < Get(k)))
            -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])))
   */

effect_6_pc by limpe ( effect_6, prime_compact_2 );
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4; dSTEP6_1, g, i,
     dSTEP6_2, f, k
   |-
   effect_6_pc:
     (@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
        ((Get(i) < Get(j) | Get(j) < Get(i)) -o
           @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
       *
       (((Get(k) < Get(i) | Get(i) < Get(k)) *
           (Get(k) < Get(j) | Get(j) < Get(k)))
```

```
          -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<* next eliminate dSTEP6_1 and dSTEP6_2 with dSTEP6_ub *>

effect_6_n1 by { mce, select} ( dSTEP6_ub, term dSTEP6_1, term dSTEP6_2,
  effect_6_pc, num 1 );
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4; g, i, f, k, dSTEP6
   |-
   effect_6_n1:
     (@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
        ((Get(i) < Get(j) | Get(j) < Get(i)) -o
           @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
       *
       (((Get(k) < Get(i) | Get(i) < Get(k)) *
           (Get(k) < Get(j) | Get(j) < Get(k)))
         -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

effect_7 by bangi ( effect_6_n1 );
/*
   dSTEP2_1, h, j, dSTEP2_2, f, k; f, k, dSTEP4; g, i, f, k, dSTEP6
   |-
   effect_7:
     !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
        ((Get(i) < Get(j) | Get(j) < Get(i)) -o
           @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
        *
        (((Get(k) < Get(i) | Get(i) < Get(k)) *
            (Get(k) < Get(j) | Get(j) < Get(k)))
          -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<*
* We could go on to try to get the effect-statement into a form
* symmetric in i and j, for example, the equivalent of
*
*    a[j] = a'[i] *
*    a[i] = a'[j] *
*    for i1: N. h1: i1 < n.
*       a[i1] = a'[i1]
*
* However, we cannot quite reach this form, but rather a form
* with these conjuncts embedded in it and some some extra copies
* of some some of the conjuncts. This is because the conjuncts
* are not individually banged. If the >>-To-* rule did not have
* the condition that A must be (passive-) discrete in A >> B
* to convert to A' * B, we wouldn't have this problem. Then
* something like (!A * !B) >> C would simply convert as
* !A' * !B' * C, which would be much more convenient. Embedding
* specification in the language has its price.
*
* We will simply leave it this way. The remaining steps are to
```

```
* remove dummies from the left context using the appropriate
* elimination rules, and thread the context. *>

effect_8 by { mce, select} ( STEP2, term dSTEP2_1, term dSTEP2_2,
  effect_7, num 1 );
/*
   h, j, f, k, temp, a, i, g; f, k, dSTEP4; g, i, f, k, dSTEP6
   |-
   effect_8:
     !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
         ((Get(i) < Get(j) | Get(j) < Get(i)) -o
            @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
        *
        (((Get(k) < Get(i) | Get(i) < Get(k)) *
            (Get(k) < Get(j) | Get(j) < Get(k)))
           -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

effect_9 by { cut, select} ( STEP4, term dSTEP4, effect_8, num 1 );
/*
   h, j, f, k, temp, a, i, g; f, k, a, j, h, i, g; g, i, f, k, dSTEP6
   |-
   effect_9:
     !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
         ((Get(i) < Get(j) | Get(j) < Get(i)) -o
            @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
        *
        (((Get(k) < Get(i) | Get(i) < Get(k)) *
            (Get(k) < Get(j) | Get(j) < Get(k)))
           -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

effect_10 by { cut, select, threading} ( STEP6, term dSTEP6, effect_9,
  num 1 );
/*
   h, j, f, k, temp, a, i, g
   |-
   effect_10:
     !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
         ((Get(i) < Get(j) | Get(j) < Get(i)) -o
            @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
        *
        (((Get(k) < Get(i) | Get(i) < Get(k)) *
            (Get(k) < Get(j) | Get(j) < Get(k)))
           -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<* abstract out the index k and its guard f *>

effect_11 by fori ( term f, effect_10 );
/*
   h, j, k, temp, a, i, g
   |-
   effect_11:
```

```
        (for f: !(Get(k) < n) .
           !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
               ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                  @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
              *
              (((Get(k) < Get(i) | Get(i) < Get(k)) *
                  (Get(k) < Get(j) | Get(j) < Get(k)))
                 -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
      */

   effect_12 by fori ( term k, effect_11 );
   /*
      h, j, temp, a, i, g
      |-
      effect_12:
        (for k: !(N) .
           for f: !(Get(k) < n) .
             !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
                 ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                    @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
                *
                (((Get(k) < Get(i) | Get(i) < Get(k)) *
                    (Get(k) < Get(j) | Get(j) < Get(k)))
                   -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
      */

 end /*STEP2_4_6*/;
 /*
    h, j, temp, a, i, g
    |-
    STEP2_4_6:
      (for k: !(N) .
         for f: !(Get(k) < n) .
           !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
               ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                  @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
              *
              (((Get(k) < Get(i) | Get(i) < Get(k)) *
                  (Get(k) < Get(j) | Get(j) < Get(k)))
                 -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
    */

 <* at long last, abstract out the temporary variable as a local *>

 result by { lve, select, contraction} ( term temp, STEP2_4_6, num 1 );
 /*
    h, j, a, i, g
    |-
    result:
      for k: !(N) .
        for f: !(Get(k) < n) .
          !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
              ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                 @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
```

```
            *
            (((Get(k) < Get(i) | Get(i) < Get(k)) *
                (Get(k) < Get(j) | Get(j) < Get(k)))
               -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
      */

end /*op*/;
/*
   h, j, a, i, g
   |-
   op:
     for k: !(N) .
       for f: !(Get(k) < n) .
         !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
             ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
           *
           (((Get(k) < Get(i) | Get(i) < Get(k)) *
               (Get(k) < Get(j) | Get(j) < Get(k)))
              -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)])
   */

<* abstract away i, g, j, h *>

op_1 by fori ( term h, op );
/*
   j, a, i, g
   |-
   op_1:
     (for h: !(Get(j) < n) .
        for k: !(N) .
          for f: !(Get(k) < n) .
            !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
                ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                   @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
              *
              (((Get(k) < Get(i) | Get(i) < Get(k)) *
                  (Get(k) < Get(j) | Get(j) < Get(k)))
                 -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
   */

op_2 by fori ( term j, op_1 );
/*
   a, i, g
   |-
   op_2:
     (for j: !(N) .
        for h: !(Get(j) < n) .
          for k: !(N) .
            for f: !(Get(k) < n) .
              !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
                  ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                     @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
                *
```

```
                 (((Get(k) < Get(i) | Get(i) < Get(k)) *
                     (Get(k) < Get(j) | Get(j) < Get(k)))
                    -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
   */

op_3 by fori ( term g, op_2 );
/*
   a, i
   |-
   op_3:
     (for g: !(Get(i) < n) .
        for j: !(N) .
          for h: !(Get(j) < n) .
            for k: !(N) .
              for f: !(Get(k) < n) .
                !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
                    ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                       @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
                   *
                   (((Get(k) < Get(i) | Get(i) < Get(k)) *
                       (Get(k) < Get(j) | Get(j) < Get(k)))
                      -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
   */

op_4 by fori ( term i, op_3 );
/*
   a
   |-
   op_4:
     (for i: !(N) .
        for g: !(Get(i) < n) .
          for j: !(N) .
            for h: !(Get(j) < n) .
              for k: !(N) .
                for f: !(Get(k) < n) .
                  !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
                      ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                         @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
                     *
                     (((Get(k) < Get(i) | Get(i) < Get(k)) *
                         (Get(k) < Get(j) | Get(j) < Get(k)))
                        -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
   */

<* and finally abstract away "a" *>

exch by fori ( term a, op_4 );
/*

   |-
   exch:
     (for a: Arrayname[N, n] .
        for i: !(N) .
          for g: !(Get(i) < n) .
```

```
          for j: !(N) .
            for h: !(Get(j) < n) .
              for k: !(N) .
                for f: !(Get(k) < n) .
                  !(@[a, Get(j), Get(h)] = @[a', Get(i), Get(g)] *
                      ((Get(i) < Get(j) | Get(j) < Get(i)) -o
                         @[a, Get(i), Get(g)] = @[a', Get(j), Get(h)]))
                     *
                     (((Get(k) < Get(i) | Get(i) < Get(k)) *
                         (Get(k) < Get(j) | Get(j) < Get(k)))
                        -o @[a, Get(k), Get(f)] = @[a', Get(k), Get(f)]))
*/
```